Preside CMS Documentation

Release 0.1.0

Pixl8 Interactive

December 11, 2015

Contents

Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

CHAPTER 1

Sticker

Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

1.1 Sticker

Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

1.1.1 Sticker

Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker

Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker

Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker Sticker is a lightweight CFML framework focused on managing CSS and JavaScript includes on a per-request basis.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

> `/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**
>
> **and**
>
> `http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
```

```
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );
```

```
        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );
```

```
        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
        bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
    sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
           .includeData( data={ resultsPerPage=5 } );
```

```
    <script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"
```

```
        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
```

```
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
        bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes**().

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include**() and **includeData**() methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
<cfset sticker.include( assetId="jquery" )
               .include( assetId="bootstrapjs" )
               .include( assetId="bootstrapcss" )
               .include( assetId="sitecss" )
               .include( assetId="sitejs" )
               .include( assetId="specific-#pageType#", throwOnMissing=false )
               .include( assetId="modernizr", group="headjs" ) />

...
```

```
    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
```

```
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );
```

```
        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
        bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
    sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
           .includeData( data={ resultsPerPage=5 } );
```

```
    <script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
```

```
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
```

```
                }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
        bundle.asset( "print-css" ).setMedia( "print" );
    }
```

```
}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
<cfset sticker.include( assetId="jquery" )
               .include( assetId="bootstrapjs" )
               .include( assetId="bootstrapcss" )
               .include( assetId="sitecss" )
               .include( assetId="sitejs" )
```

```
                    .include( assetId="specific-#pageType#", throwOnMissing=false )
                    .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
```

```
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {
```

```
    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
        bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {
```

```
        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
```

```
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {
```

```
    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
    sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
           .includeData( data={ resultsPerPage=5 } );
```

```
    <script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
        bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
```

```
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell

Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker

Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker

The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets

Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes

Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

Sticker in a nutshell

Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
<cfset sticker.include( assetId="jquery" )
               .include( assetId="bootstrapjs" )
               .include( assetId="bootstrapcss" )
               .include( assetId="sitecss" )
               .include( assetId="sitejs" )
               .include( assetId="specific-#pageType#", throwOnMissing=false )
               .include( assetId="modernizr", group="headjs" ) />

...
```

```
    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

Installing Sticker

Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

Starting up Sticker

The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets

Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
```

```
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies

By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       // the sitecore asset depends on jquery, all other assets depend on sitecore
       bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

       // the core css file should come before all others
       bundle.asset( "core-css" ).before( "*" );

       // the blog-template css file should come after 'common-css' and 'social-css'
       bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media

You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes

Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

Declare that you wish to include a given asset when you later call **renderIncludes()**.

The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

1.1.2 Sticker in a nutshell

Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

1.1.3 Installing Sticker

Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

1.1.4 Starting up Sticker

The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

1.1.5 Configuring your assets

Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
```

```
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
              }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

Specifying sort order and dependencies

By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );
```

```
        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

Specifying IE restrictions and CSS media

You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
        bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

1.1.6 Including assets in your request and rendering includes

Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes**().

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
    sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
           .includeData( data={ resultsPerPage=5 } );
```

```
    <script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

1.2 Sticker in a nutshell

Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

1.3 Installing Sticker

Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

1.4 Starting up Sticker

The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"               , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

1.5 Configuring your assets

Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
```

```
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

1.5.1 Specifying sort order and dependencies

By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }
```

```
}
```

1.5.2 Specifying IE restrictions and CSS media

You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

1.6 Including assets in your request and rendering includes

Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
    sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
           .includeData( data={ resultsPerPage=5 } );
```

```
    <script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include**() and **includeData**() methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.

CHAPTER 2

Sticker in a nutshell

Sticker helps you out by mapping your static resources to useful IDs. These IDs can then be referenced in your CFML code. So:

`/js/lib/2bf82ac6-sitecore.min.js` becomes **"sitecore"**

and

`http://cdn.jquery.com/jquery-34.25.34.min.js` becomes **"jquery"**

A typical layout template that uses Sticker might look like this:

```
    <cfset sticker.include( assetId="jquery" )
                  .include( assetId="bootstrapjs" )
                  .include( assetId="bootstrapcss" )
                  .include( assetId="sitecss" )
                  .include( assetId="sitejs" )
                  .include( assetId="specific-#pageType#", throwOnMissing=false )
                  .include( assetId="modernizr", group="headjs" ) />

    ...

    #sticker.renderIncludes( type="css" )#
    #sticker.renderIncludes( group="headjs" )#
</head>
<body>
    ...

    #sticker.renderIncludes( type="js" )#
</body>
```

CHAPTER 3

Installing Sticker

Sticker can be downloaded from Forgebox. Once downloaded and unpacked, create a mapping to the sticker directory called **'/sticker'** (not required if you unpacked sticker to the webroot).

CHAPTER 4

Starting up Sticker

The Sticker API is designed to be a Singleton and any instances you create should be cached in a permanent scope, e.g. the application scope. An example instantiation, using `Application.cfc`, might look like this:

```
component {
    //...
    function onApplicationStart() {

        // 1. instantiate sticker with no arguments
        var sticker = new sticker.Sticker();

        // 2. add bundles, each bundle is simply a folder containing static assets
        //    and must have a StickerBundle.cfc file in it's root directory to set its configuration
        sticker.addBundle( rootDirectory="/assets"                , rootUrl="http://mywebsite-static.c
               .addBundle( rootDirectory="/myCompanyCoreAssetLib", rootUrl="/corelib/"

        // 3. call load(), this will read all the bundles and merge their definitions
        sticker.load();

        application.sticker = sticker;
    }
}
```

Configuring your assets

Sticker uses `StickerBundle.cfc` configuration files that are planted in the root of your static asset folders. You might, for example, have a folder structure like this:

```
/wwwroot
    /assets
        /js
        /css
        /images
        StickerBundle.cfc
```

And a `StickerBundle.cfc` file that looks like this:

```
component {

    // all valid StickerBundle.cfc files must implement the 'configure()' method
    public void function configure( bundle ) {
        // registering a single, remote asset
        bundle.addAsset( id="jquery", url="http://cdn.jquery.com/jquery-34.25.34.min.js" );

        // registering a single, local asset
        // note the wildcard filename map to help with cachebusters in the filename.
        bundle.addAsset( id="sitecore", path="/js/*-sitecore.min.js" );

        // registering multiple assets at once
        // notice the idGenerator closure function that can be used to format your asset IDs
        // based on each matched asset
        bundle.addAssets(
              directory   = "/css"
            , filter      = "*.min.css"
            , idGenerator = function( filePath ){
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.css$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );

        // same as above, but using a function for the filter
        bundle.addAssets(
              directory   = "/js"
            , filter      = function( filePath ){ return ReFindNoCase( "\.min\.js$", filePath ); }
            , idGenerator = function( filePath ){
```

```
                var id = Replace( filepath, "/", "-", "all" );
                id = ReReplace( filePath, "\.min\.js$", "" );
                id = ReReplace( filePath, "^-", "" );

                return id;
            }
        );
    }

}
```

Note: All paths in your `StickerBundle.cfc` file are **relative** to the parent directory of the `StickerBundle.cfc`

5.1 Specifying sort order and dependencies

By default, your assets will be rendered in alphabetical order. However, you can define sort orders and dependencies by modifying your `StickerBundle.cfc`, using the following methods:

- `before()`
- `after()`
- `dependsOn()`
- `dependents()`

Example:

```
component {

    public void function configure( bundle ) {
        // etc...

        // the sitecore asset depends on jquery, all other assets depend on sitecore
        bundle.asset( "sitecore" ).dependsOn( "jquery" ).dependents( "*" );

        // the core css file should come before all others
        bundle.asset( "core-css" ).before( "*" );

        // the blog-template css file should come after 'common-css' and 'social-css'
        bundle.asset( "blog-template-css" ).after( "common-css", "social-css" );

    }

}
```

5.2 Specifying IE restrictions and CSS media

You may wish to conditionally include a file for IE 7 and below, or specify that a stylesheet is only for *print*. To do so, you can make use of the following methods:

- `setIE()`
- `setMedia()`

Example:

```
component {

    public void function configure( bundle ) {
       // etc...

       bundle.asset( "hacky-ie-js" ).setIE( "lte IE 7" );
       bundle.asset( "print-css" ).setMedia( "print" );
    }

}
```

Including assets in your request and rendering includes

Once your Sticker instance is configured and loaded, all you have left to do is include assets and render them in your page. There are three methods available:

- `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`
- `sticker.includeData( required struct data, string group="default" )`
- `sticker.renderIncludes( string type, string group="default" )`

See an explanation of each below:

1. `sticker.include( required string assetId, boolean throwOnError=true, string group="default" )`

   Declare that you wish to include a given asset when you later call **renderIncludes()**.

   The *throwOnError* argument is useful when you wish to include assets by convention and don't want to see an error when they are not found.

   The *group* argument can be used for when you need to render multiple groups of includes, i.e. some JS might be required to live in the HTML head, while the rest belongs at the end of the body

2. `sticker.includeData( required struct data, string group="default" )`

   Include data allows you to make some server-side data available to your javascript. See the following call and output for an illustration:

```
sticker.includeData( data={ lookupUrl="http://ajax.mysite.com/lookup/" } )
       .includeData( data={ resultsPerPage=5 } );
```

```
<script>var cfrequest={"lookupUrl":"http://ajax.mysite.com/lookup/","resultsPerPage":5}</script>
```

3. `sticker.renderIncludes( string type, string group="default" )`

   Renders any includes and data that have been specified with the **include()** and **includeData()** methods.

   If you do not specify the *type* argument, the method will render any CSS followed by JS. Otherwise, you may pass either "css" or "js".

   If you specify the *group* argument, only assets and data included with the same group name will be rendered.