generator-politico-django Documentation

Release 0.2.0

Jon McClure

Feb 03, 2019

Contents:

A Yeoman generator for building a modern front-end development environment inside a Django application.

It uses Webpack, Babel and SASS.

CHAPTER 1

Why this?

At POLITICO, Django is our choice backend, but developing front-end assets in a Python framework isn't always the best experience. For example, to take advantage of the latest JavaScript frameworks, we want to use the latest JS build tools.

This generator helps incorporate a node-based development environment inside a Django application. It creates a build system that integrates directly with Django's normal static files handling pattern while letting you use robust JavaScript module patterns, the latest syntax and prepocessors.

We've found it helps keep our apps better organized and our front-end code as clean and modular as our Python in a way that doesn't require any Django gymnastics.

CHAPTER 2

Installing

2.1 Dependencies

Make sure you have the latest version of node installed on your machine as well as the yarn package manager.

2.2 NPM

Install the package's dependencies globally.

```
$ npm install -g gulp-cli yo generator-politico-interactives
```

Install the package globally.

```
$ npm install -g generator-politico-django
```

2.3 Symlink

Alternatively, you can clone a copy of the generator's git repository and use a symlink to install the package. This is especially useful if you'll be developing the generator.

```
$ git clone git@github.com:The-Politico/generator-politico-django.git

$ cd generator-politico-django

$ npm link
```

Note: To update a symlinked package, just `git pull` the latest changes in the symlinked directory.

CHAPTER 3

Using

3.1 Concepts

The generator builds the structure for a lightweight application that will compile your static assets using Webpack.

In development, the app will proxy Django's development server, serve your static files and push changes to your browser using Hot Module Replacement.

After you're done developing, the app will build your static assets, which will minify scripts and styles and move them to the normal static directory of your app, i.e., `js/` and `css/` directories.

Warning: This app presumes your static directory is structured in the standard Django way using namespacing. For example: `<your app>/static/<your app>/js/`

3.2 Setup

1. Within your Django application, make a new folder that will contain your raw assets.

```
$ mkdir staticapp
$ cd staticapp
```

Note: You can name the directory whatever, but we'll assume you've called it `staticapp` throughout these docs.

2. Run the generator.

```
$ yo politico-django
```

3.3 Developing assets

To begin developing assets, simply start the development pipeline in your `staticapp` directory.

```
$ yarn start
```

During development, the app will proxy Django's own default web server, i.e., `runserver`. You can customize the proxied port number using the `proxy` argument, which defaults to `8000`.

```
$ yarn start --proxy 8008
```

3.3.1 JavaScript

Write your scripts using modern ES2015 syntax. Babel transforms for React/JSX are also included by default.

In order to build separate scripts for different views in your app, Webpack will look for bundle entries using a glob pattern `main.*.js*`. So simply prefix any JS or JSX files with `main.` to create a new bundle at the root of your `src/` directory.

For example, these scripts will be compiled into a single bundle, `main.app.js`:

```
// abide.js

export default (name) => {
  console.log(`The ${name} abides!`)
};
```

```
// main-app.js

import Abide from './abide';

Abide('Dude');
```

3.3.2 SCSS

Import SCSS files in your JavaScript.

```
import './theme/base.scss';;
```

As per the POLITICO JS Apps Style Guide, SCSS files outside your `theme/` directory will be imported as CSS modules.

Fig. 1: You got styles in my scripts!

3.3.3 Django templates

In your Django templates, you can reference scripts and styles using Django's static files template tag.

```
{% load static %}

<link rel="stylesheet" href="{% static '<your app>/css/main.app.css' %}" />

<script src="{% static '<your app>/js/main.app.js' %}"></script>
```

In development, the development server will serve your JavaScript modules at the location of your app's static directory. For example: `localhost:3000/static/myapp/js/main.app.js`.

Your styles will be delivered within your JavaScript bundle and injected onto the page. This lets Webpack automatically refresh your styles as you develop.

Note: Because the proxy server serves your styles via JavaScript in development, you should see a 404 error in your template for your link tag.

When you build your scripts for production, the styles will be split into a separate file **named after your module.** For example, a module named `main.app.js` will split CSS styles to a stylesheet at `css/main.app.css`.

Warning: If you build your static assets and then return to using the development server, keep in mind, that your previously built styles may be included in your template. So using the above example, a stale `main.app.css` may be referenced in your template.

If you're simply overwriting styles, the new styles will be injected after the reference to the stale built asset and shouldn't cause a problem, but any other style conflicts may show through.

Best practice if you're revisiting assets is to delete the stale built files from your app's static directory.

Note: Any changes you make to JavaScript or SCSS files will be automatically reflected in your browser via Hot Module Replacement. Any changes you make in your template's HTML or your Django view will still require you to refresh your browser.

3.4 Building production assets

Once you've finished developing assets. Run Gulp's build task inside your `staticapp` directory:

```
$ yarn build
```

This will minify your bundles, separate CSS bundles and move scripts and stylesheets to your app's static files folder.