Awe Documentation

Release 1.1.0

Dave James Miller

24 July 2015

Contents

1 Introduction **1**
1.1 Features . . . 1

2 Installation **3**
2.1 Quick start . . . 3
2.2 System requirements . . . 3
2.3 Installing . . . 5
2.4 Upgrading . . . 5
2.5 Uninstalling . . . 6

3 Project configuration files **7**
3.1 awe.yaml . . . 7
3.2 About the YAML format . . . 7
3.3 Config sections . . . 8

4 Asset building **11**
4.1 Getting started . . . 11
4.2 Autoprefixer . . . 12
4.3 CoffeeScript . . . 13
4.4 Sass . . . 13
4.5 Ignored files (partials) . . . 13
4.6 Compass . . . 14
4.7 Sprites . . . 15
4.8 Combining files . . . 16
4.9 Import files . . . 17
4.10 Bower support . . . 17
4.11 Multiple asset groups . . . 19

5 Using with CMSs and frameworks **21**
5.1 WordPress . . . 21
5.2 Laravel 5 . . . 22
5.3 Laravel 4 . . . 22

6 Cache files **23**
6.1 Hiding in Sublime Text . . . 23

7 Upgrading a project **25**
7.1 v0.1.0 (16 Nov 2014) . . . 25

8 Quick reference **27**
8.1 Command-line interface (`awe`) . . . 27
8.2 Configuration file (`awe.yaml`) . . . 27
8.3 Assets directory structure . . . 28
8.4 YAML import files . . . 29

9 Design decisions **31**
9.1 Introduction . . . 31
9.2 Specific- not general-purpose . . . 31
9.3 System-wide installation . . . 31
9.4 Unit tests . . . 32
9.5 Conservative defaults . . . 32
9.6 Minimal configuration . . . 32
9.7 YAML configuration . . . 32
9.8 Automatic mapping of asset files . . . 32
9.9 YAML import files . . . 33
9.10 No shorthand syntax in import files . . . 33
9.11 Limited file type support . . . 33
9.12 Open source . . . 33
9.13 Flag deprecated features . . . 33
9.14 Runs in a terminal (SSH)... . . . 34
9.15 Both asset building *and* deployment . . . 34

10 Contributing **35**
10.1 Introduction . . . 35
10.2 System requirements . . . 35
10.3 Installing Awe from Git . . . 36
10.4 Source code . . . 37
10.5 Unit tests . . . 38
10.6 Documentation . . . 38
10.7 Updating dependencies . . . 40
10.8 Releasing a new version . . . 40

CHAPTER 1

Introduction

Awe is a tool used at Alberon[1] to simplify the building and maintenance of websites / web apps, by handling the compilation of assets. It makes it easy to compile CoffeeScript & Sass files, autoprefix CSS files and combine source files together, with full source map support for easier debugging.

(In the future it will also handle deployment to remote live/staging servers, and related functionality such as configuration management.)

While it is not designed to be used by third parties, it is open source and you're welcome to use it if you want to!

1.1 Features

1.1.1 Assets

- Compile CoffeeScript[2] (`.coffee`) files to JavaScript
- Compile Sass[3]/Compass[4] (`.scss`) files to CSS
- Autoprefix[5] CSS rules for easier cross-browser compatibility
- Combine multiple JavaScript/CSS source files into a single file
- Rewrite relative URLs in CSS files that are combined (e.g. packages installed with Bower[6])
- Generate source maps[7] to aid debugging
- Watch for changes to source files and rebuild automatically

1.1.2 Deployment

Coming soon!

[1] http://www.alberon.co.uk
[2] http://coffeescript.org/
[3] http://sass-lang.com/
[4] http://compass-style.org/
[5] https://github.com/ai/autoprefixer
[6] http://bower.io/
[7] http://www.html5rocks.com/en/tutorials/developertools/sourcemaps/

CHAPTER 2

Installation

Alberon Note

If you are using Jericho (Alberon's shared development server), Awe is already installed and you can skip to Project configuration files (page 7).

2.1 Quick start

If you already know what you're doing, this is a shorthand version of the instructions below for Debian Wheezy:

```
$ echo 'deb     http://ftp.uk.debian.org/debian/ wheezy-backports main contrib non-free' | sudo tee -
$ sudo apt-get update
$ sudo apt-get install curl nodejs nodejs-legacy ruby ruby-dev
$ curl https://www.npmjs.org/install.sh | sudo sh
$ sudo gem install bundler
$ sudo npm install -g awe bower
```

2.2 System requirements

2.2.1 Linux

Awe is developed and tested on Linux. It should run on Mac OS X too, but it hasn't been tested. It probably won't work on Windows (at least not 100%) because it uses symlinks.

Future Plans

I could add Windows support if there is demand for it, but this would add some complexity (e.g. symlinks to `bower_components/` would not be possible so the files would need to be copied instead).

2.2.2 Node.js & npm

Node.js[1] and npm[2] must be installed. Awe is tested on Node.js 0.10, and may not work on older versions.

To check they're installed, run:

[1] https://nodejs.org/
[2] https://www.npmjs.org/

```
$ node --version
$ npm --version
```

Installing Node.js on Debian

On Debian Wheezy, Node.js is only available in Backports[3], but you will need to add that as a source:

```
$ sudo vim /etc/apt/sources.list
```

Add this line:

```
deb     http://ftp.uk.debian.org/debian/ wheezy-backports main contrib non-free
```

Then run the following to install it:

```
$ sudo apt-get update
$ sudo apt-get install curl nodejs nodejs-legacy
$ curl https://www.npmjs.org/install.sh | sudo sh
```

2.2.3 Ruby & Bundler

You must also have Ruby[4] and Bundler[5] installed - they are required to run Compass[6], Sass files to CSS.

Since Awe is installed system-wide, they also need to be installed system-wide - i.e. not using RVM[7] or rbenv[8].

To check they're installed, run:

```
$ ruby --version
$ bundle --version
```

(Compass itself will be installed by Awe, so it does not need to be installed manually.)

Installing Ruby on Debian

```
$ sudo apt-get install ruby ruby-dev
$ sudo gem install bundler
```

2.2.4 Bower (optional)

You may also install Bower[9] for managing third-party assets:

```
$ sudo npm install -g bower
```

To check it's installed, run:

```
$ bower --version
```

[3] http://backports.debian.org/
[4] https://www.ruby-lang.org/
[5] http://bundler.io/
[6] http://compass-style.org/
[7] https://rvm.io/
[8] https://github.com/sstephenson/rbenv
[9] http://bower.io/

2.3 Installing

Simply install Awe using npm:

```
$ sudo npm install -g awe
```

This will install the Awe package globally, including the `awe` executable, and also download the Node.js and Ruby dependencies.

To check it's installed, run:

```
$ awe --version
```

2.3.1 Installing a specific version

To install a specific version, use the `awe@<version>` syntax of npm, for example:

```
$ sudo npm install -g awe@1.0.0
```

To see a list of all available versions, see the list of releases[10] or the list of commits[11].

2.4 Upgrading

Because Awe is installed globally, you only need to upgrade it once per machine, not separately for each project. Every effort will be made to ensure backwards compatibility, though you should check Upgrading a project (page 25) to see if anything important has changed.

2.4.1 Checking for updates

```
$ npm outdated -g awe
```

If Awe is up to date, only the headings will be displayed:

```
Package  Current  Wanted  Latest  Location
```

If there is a newer version, the currently installed version and latest version number will be displayed:

```
Package  Current  Wanted  Latest  Location
awe        1.0.0   1.1.0   1.1.0  /usr/lib > awe
```

2.4.2 Upgrading to the latest version

```
$ sudo npm update -g awe
```

[10] https://github.com/alberon/awe/releases
[11] https://github.com/alberon/awe/commits

2.4.3 Upgrading to a specific version

To upgrade (or downgrade) to a specific version, use `install` instead:

```
$ sudo npm install -g awe@1.0.0
```

2.5 Uninstalling

To remove Awe from your machine, simply uninstall it with npm:

```
$ sudo npm uninstall -g awe
```

This will also delete the Node.js and Ruby dependencies that were downloaded automatically during installation (e.g. CoffeeScript, Sass, Compass). It will not remove any project files (configuration, cache files or compiled assets).

CHAPTER 3

Project configuration files

3.1 awe.yaml

Each project requires a single config file, `awe.yaml`, in the root directory.

A new config file can be created in the current directory by running `awe init`:

```
$ cd /path/to/repo
$ awe init
```

Then simply open `awe.yaml` in your preferred text editor to customise it as needed.

Tip: If you prefer you can create a config file by hand, or copy one from another project – but I recommend using `awe init` to ensure you're starting with the latest recommended settings.

3.2 About the YAML format

The file is in YAML[1] format. This is similar in purpose to JSON, but easier to read and write. Here is an example config file in YAML:

```
# Awe config - see http://awe.alberon.co.uk/ for documentation

ASSETS:

    # This is a comment
    default:
        src:        www/wp-content/themes/mytheme/src/
        dest:       www/wp-content/themes/mytheme/build/
        bower:      false
        sourcemaps: true
```

Note how indentation is used to determine the structure, similar to Python and CoffeeScript, and strings do not need to be quoted. It also supports real comments, unlike JSON.

You shouldn't need to understand YAML in detail to configure Awe – just follow the examples – but if you would like to learn more about it please see Wikipedia[2] or the official YAML specification[3].

[1] http://yaml.org/
[2] http://en.wikipedia.org/wiki/YAML
[3] http://www.yaml.org/spec/1.2/spec.html#Preview

Note: For comparison, the equivalent config file in JSON would be:

```
{
    "_comment": "Awe config - see http://awe.alberon.co.uk/ for documentation",

    "ASSETS": {

        "_comment": "This is a comment (http://stackoverflow.com/a/244858/167815)",
        "theme": {
            "src":         "www/wp-content/themes/mytheme/src/",
            "dest":        "www/wp-content/themes/mytheme/build/",
            "bower":       false,
            "sourcemaps": true
        }

    }
}
```

But this is just an illustration – JSON is *not* supported by Awe.

3.3 Config sections

The config file is designed to be split into sections. Each top-level section is written in UPPERCASE to make it stand out:

```
ASSETS:

    # Asset groups config
```

For more information about the settings available, see:

- Asset building (page 11)
- *Quick reference* (page 27)

Future Plans

Currently the only section supported is `ASSETS`, but in the future the config file may look something like this:

```
ASSETS:

    # Asset groups config

CONFIG:

    # Custom settings

CRON:

    # Cron jobs config

DEPLOY:

    # Deployment config

ENVIRONMENTS:

    # Configure environments (dev, staging, live)
```

```
MYSQL:

    # MySQL config

PERMISSIONS:

    # File permissions config

SETUP:

    # Setup command config (e.g. npm, composer, bundler)

VERSIONS:

    # Require specific versions of Awe, CoffeeScript, etc.
```

CHAPTER 4

Asset building

4.1 Getting started

4.1.1 Create your source directory

First, create a directory for your source files. Let's say you're making a WordPress[1] theme - you would create a subdirectory named `src/` in your theme as follows:

```
$ mkdir www/wp-content/themes/mytheme/src/
```

4.1.2 Configuration

Next, add the following to the `awe.yaml` configuration file (page 7), replacing the paths as necessary:

```
ASSETS:

    default:
        src:          www/wp-content/themes/mytheme/src/
        dest:         www/wp-content/themes/mytheme/build/
        bower:        false
        autoprefixer: false
```

Warning: The `build/` directory **should not** be an existing directory – anything inside it will be deleted.

Tip: The `src/` directory can be outside the document root if you prefer. e.g. The recommended configuration for Laravel[2] 5 is:

```
ASSETS:

    default:
        src:          resources/assets/     # app/assets/ in Laravel 4
        dest:         public/assets/
```

Be aware that the original source code will still be made public (in the source maps), so this is not a way to hide it.

[1] https://wordpress.org/
[2] http://laravel.com/

4.1.3 Create your source files

All your source files should go into the `src/` directory you created above. For now, let's imagine you have these files:

```
src/
+-- img/
|   +-- logo.png
+-- sample1.css
+-- sample2.js
+-- subdirectory/
    +-- A.css
    +-- B.js
```

4.1.4 Run the build command

Finally, run the `build` command to generate the `build/` directory:

```
$ awe build
```

Or run the `watch` command to generate it and then wait for further changes:

```
$ awe watch
```

Since there are no special files in the list above, you will get exactly the same structure:

```
build/
+-- img/
|   +-- logo.png
+-- sample1.css
+-- sample2.js
+-- subdirectory/
    +-- A.css
    +-- B.js
```

However, read on to see what Awe can do!

4.2 Autoprefixer

Autoprefixer[3] automatically adds vendor prefixes (`-webkit-`, `-moz-`, etc.) to your CSS files. Simply enable it in the config:

```
ASSETS:

    default:
        src:          www/wp-content/themes/mytheme/src/
        dest:         www/wp-content/themes/mytheme/build/
        bower:        false
        autoprefixer: true
```

For more details about how it works, and how to selectively disable it, see the Autoprefixer documentation[4].

[3] https://github.com/postcss/autoprefixer
[4] https://github.com/postcss/autoprefixer#readme

4.3 CoffeeScript

CoffeeScript[5] is "a little language that compiles into JavaScript". It has a very simple 1-to-1 mapping of input files (`.coffee`) to output files (`.js`). For example, these source files:

```
src/
+-- sample.coffee
+-- subdirectory/
    +-- A.coffee
```

Would result in this output:

```
build/
+-- sample.js
+-- subdirectory/
    +-- A.js
```

Tip: It will also generate source maps – `sample.js.map` and `subdirectory/A.js.map` – but these are not shown for simplicity.

For more details see the CoffeeScript documentation[6].

4.4 Sass

Sass[7] is an extension to CSS, and compiles `.scss` files to `.css`. For example, these source files:

```
src/
+-- sample.scss
+-- subdirectory/
    +-- A.scss
```

Would result in this output:

```
build/
+-- sample.css
+-- subdirectory/
    +-- A.css
```

For more details see the Sass documentation[8].

Note: Only the *SCSS* format is supported by Awe, not the original *Sass* indented format (i.e. `.sass` files), because it's easier for people used to regular CSS to pick up.

4.5 Ignored files (partials)

Awe ignores all files and directories that start with an underscore (_). In Sass this is used to `@import` partials[9] – for example, this directory structure:

[5] http://coffeescript.org/
[6] http://coffeescript.org/
[7] http://sass-lang.com/
[8] http://sass-lang.com/guide
[9] http://sass-lang.com/guide#topic-4

```
src/
+-- _partials/
|   +-- reset.scss
+-- _vars.scss
+-- styles.scss
```

Will result in this output:

```
build/
+-- styles.css
```

Note: Although this is mostly used for Sass partials, Awe will ignore **any** file or directory that starts with an underscore.

4.6 Compass

Compass[10] is a popular CSS framework built on top of Sass. To use it, simply `@import` the file shown in the Compass documentation[11] at the top of your `.scss` file. For example:

```
@import 'compass/typography/links/unstyled-link';

.footer a {
    @include unstyled-link;
}
```

This is compiled to:

```
.footer a {
    color: inherit;
    text-decoration: inherit;
    cursor: inherit;
}

.footer a:active, .footer a:focus {
    outline: none;
}
```

Tip: While it is possible to use `@import 'compass';` as a short-hand, this is noticably slower to build than importing only the specific features required.

Tip: Many of the Compass mixins simply add vendor prefixes for CSS3[12]. Instead of using these, I recommend enabling *autoprefixer* (page 12).

Note: You may need to be aware of the following Compass configuration options[13] that Awe uses:

```
images_path      = 'src/img/'                    # used by image-url(), inline-image(), etc.
fonts_path       = 'src/fonts/'                  # used by font-url(), inline-font-files(), etc.
sprite_load_path = ['src/img/', 'src/_sprites/']  # used for sprite generation (see below)
```

[10] http://compass-style.org/
[11] http://compass-style.org/reference/compass/
[12] http://compass-style.org/reference/compass/css3/
[13] http://compass-style.org/help/documentation/configuration-reference/

This means images should be kept in a folder called `img/`, font files in `fonts/` and sprites in `_sprites/`.

4.7 Sprites

Compass has the ability to take several small icons and combine them into a single image, then use that as a sprite in your CSS.

To do this, first create a directory inside `src/_sprites/` with the name of the sprite – e.g. `src/_sprites/navbar/`. Inside that directory create a PNG image for each icon. You can also have variants ending with `_hover`, `_active` and `_target` which map to `:hover`, `:active` and `:target` in the CSS. So, for example, you may have a directory structure like this:

```
src/
+-- _sprites/
|   +-- navbar/
|       +-- edit.png
|       +-- edit_hover.png
|       +-- ...
|       +-- save.png
|       +-- save_hover.png
+-- sample.scss
```

Then in the SCSS file enter the following:

```
@import 'compass/utilities/sprites';
@import 'navbar/*.png';              // This path is relative to the _sprites/ directory
@include all-navbar-sprites;         // Replace 'navbar' with the directory name
```

This will generate a directory structure similar to the following:

```
build/
+-- _generated/
|   +-- navbar-s71af1c7425.png
+-- sample.css
```

And the following classes will appear in the output file, ready for you to use in your HTML:

```
/* Replace 'navbar' with the directory name */
.navbar-delete       { ... }
.navbar-delete:hover { ... }
.navbar-edit         { ... }
.navbar-edit:hover   { ... }
.navbar-new          { ... }
.navbar-new:hover    { ... }
.navbar-save         { ... }
.navbar-save:hover   { ... }
```

4.7.1 Advanced spriting

If you require more control over the classes that are generated, there are several other ways to create them. For example:

```
@import 'compass/utilities/sprites';

$navbar-map: sprite-map('navbar/*.png');
```

```
.navbar {
    background: $navbar-map;
}

@each $sprite in sprite-names($navbar-map) {
    .navbar-#{$sprite} {
        @include sprite($navbar-map, $sprite, true);
    }
}
```

For more details, please see the Compass spriting documentation[14], options[15] and mixins[16].

Note: The Compass documentation uses `images/` as the base directory, whereas Awe recommends using `_sprites/`. You can also put them in the `img/` directory if you prefer, but in that case the source images will be copied to the build directory as well.

4.8 Combining files

Awe can automatically combine multiple CSS/JavaScript files into a single file, allowing you to split the source files up neatly while reducing the number of downloads for end users.

Simply create a directory with a name that ends `.css` or `.js` and all the files within that directory will be concatenated (in alphabetical/numerical order) into a single output file. For example:

```
src/
+-- combined.css/
    +-- 1.css
    +-- 2/
    |   +-- A.css
    |   +-- B.scss
    +-- 3.scss
```

First the `.scss` files will be compiled to CSS, then all 4 files will be combined (in the order `1.css`, `2/A.css`, `2/B.scss`, `3.scss`) into a single `combined.css` file:

```
build/
+-- combined.css
```

Simple as that!

Caution: It is best to avoid mixing subdirectories and files, as some programs display all subdirectories first which may be confusing:

- `subdirectory/` (2)
- `file.css` (1)
- `vendor.css` (3)

[14] http://compass-style.org/help/tutorials/spriting/
[15] http://compass-style.org/help/tutorials/spriting/customization-options/
[16] http://compass-style.org/reference/compass/utilities/sprites/base/

4.9 Import files

Another way to combine multiple files is to create an import file – this is a YAML file with the extension `.css.yaml` or `.js.yaml` containing a list of files to import. This is mostly useful for importing vendor files:

```
src/
+-- vendor.js.yaml

vendor/
+-- chosen.js
+-- jquery.js
```

Where `vendor.js.yaml` contains:

```
- ../vendor/jquery.js
- ../vendor/chosen.js
```

Will compile to:

```
build/
+-- vendor.js
```

To import files from Bower (*see below* (page **??**)), simply prefix the filename with `bower:`:

```
- bower: jquery/jquery.js
- bower: jquery-ui/ui/jquery-ui.js
```

4.10 Bower support

Bower[17] is a package manager for third-party assets. It makes it easier to install and upgrade frontend dependencies such as jQuery and Bootstrap.

4.10.1 Create bower.json

Make sure you have a `bower.json` file – if not, run this to create one:

```
$ cd /path/to/repo
$ echo '{"name":"app","private":true}' > bower.json
```

Future Plans

I plan to add a command to generate this file, e.g. `awe init bower`, because `bower init` asks far more questions than are necessary!

4.10.2 Find packages

To find a package on Bower, run:

```
$ bower search <name>
```

Or use the online package search[18].

[17] http://bower.io/
[18] http://bower.io/search/

4.10.3 Install the packages you want

To install a package, run this:

```
$ bower install --save <name>
```

Sometimes you may need to specify a version number – e.g. jQuery will default to the 2.x branch which does not support IE8:

```
$ bower install --save jquery#1.x
```

This will create a `bower_components/` directory in the project root (same directory as `awe.yaml`) containing the package and any dependencies.

Tip: If the package you want is not registered with Bower, you can install it from another source:

```
$ bower install --save user/repo                          # From GitHub
$ bower install --save http://example.com/script.js       # From a URL
$ bower install --save http://example.com/package.zip     # From a zip
```

For more details, please see the Bower install documentation[19].

Note: The installed packages should be checked into the Git repository, not ignored, to ensure the same version is installed on the live site. This advice may change in the future when bower.lock[20] is implemented (and/or `awe deploy` is ready).

4.10.4 Update the config file

Update `awe.yaml` with the path to the Bower components directory:

```
ASSETS:

    default:
        src:          www/wp-content/themes/mytheme/src/
        dest:         www/wp-content/themes/mytheme/build/
        bower:        bower_components/
        autoprefixer: false
```

4.10.5 Import the files you need

Create a `.js.yaml` or `.css.yaml` *import file* (page 17) (e.g. `src/jquery.js.yaml`), for example:

```
- bower: jquery/jquery.js
```

This will be compiled to `build/jquery.js`.

Note: An alternative is to load the file you need directly in your HTML, using the `_bower/` symlink that is created:

```
<script src="/assets/_bower/jquery/jquery.min.js"></script>
```

---

[19] http://bower.io/docs/api/#install
[20] https://github.com/bower/bower/pull/1592

4.10.6 Combining Bower and non-Bower files

You can easily combine Bower files with custom files, as described above. For example:

```
src/
+-- app.css/
|   +-- 1-import.css.yaml    ==>    - bower: jquery-ui/themes/smoothness/jquery-ui.css
|   +-- 2-custom.scss
+-- app.js/
    +-- 1-import.js.yaml     ==>    - bower: jquery/jquery.js
    |                               - bower: jquery-ui/ui/jquery-ui.js
    +-- 2-custom.coffee
```

Will result in:

```
build/
+-- _bower/  ->  ..../bower_components/
+-- app.css
+-- app.js
```

(-> indicates a symlink.)

The URLs from `jquery-ui.css` (now in `app.css`) will automatically be rewritten to `url(_bower/jquery-ui/themes/smoothness/<filename>)`.

4.10.7 Updating packages

To check for outdated dependencies:

```
$ bower list
```

To update them, first update `bower.json` if necessary (if you have specified a particular version to use), then run:

```
$ bower update
```

For more details, please see the Bower documentation[21].

4.11 Multiple asset groups

To compile assets in multiple directories, simply add another group with a different name:

```
ASSETS:

    theme:
        src:          www/wp-content/themes/mytheme/src/
        dest:         www/wp-content/themes/mytheme/build/
        bower:        false
        autoprefixer: false

    plugin:
        src:          www/wp-content/plugins/myplugin/src/
        dest:         www/wp-content/plugins/myplugin/build/
        bower:        false
        autoprefixer: true
```

[21] http://bower.io/docs/api/

Reasons to do this include:

- Multiple themes/plugins in a single project
- Different config settings for different assets
- Speed up `watch` builds by only rebuilding one directory at a time

The group name must be alphanumeric (`[a-zA-Z0-9]+`).

Future Plans

The group name is not currently used anywhere, but in the future it may be possible to build individual directories (e.g. `awe build theme`).

CHAPTER 5

Using with CMSs and frameworks

5.1 WordPress

The recommended directory structure for WordPress is:

```
repo/
+-- bower_components/               # Bower packages
+-- www/
|   +-- wp-content/
|       +-- themes/
|           +-- mytheme/
|               +-- build/          # Build files
|               +-- src/            # Source files
|               |   +-- main.css/
|               |   +-- main.js/
|               +-- style.css       # Theme config (no CSS code!)
+-- awe.yaml                        # Awe config
```

With the following configuration:

```
ASSETS:

    default:
        src:          www/wp-content/themes/mytheme/src/
        dest:         www/wp-content/themes/mytheme/build/
        bower:        bower_components/
        autoprefixer: true
```

`style.css` should only contain the file header[1] that WordPress requires – for example:

```
/*
Theme Name: My Theme
...
*/
```

Then `main.css` should be used in the HTML code (instead of `bloginfo('stylesheet_url')`):

```
<link rel="stylesheet" href="<?= get_template_directory_uri() ?>/build/main.css">
```

[1] http://codex.wordpress.org/File_Header

5.2 Laravel 5

The recommended directory structure for Laravel[2] 5 is:

```
repo/
+-- app/
+-- bower_components/   # Bower packages
+-- public/
|   +-- assets/         # Build files
+-- resources/
|   +-- assets/         # Source files
|       +-- main.css/
|       +-- main.js/
+-- awe.yaml            # Awe config
```

With the following configuration:

```
ASSETS:

    default:
        src:          resources/assets/
        dest:         public/assets/
        bower:        bower_components/
        autoprefixer: true
```

5.3 Laravel 4

The recommended directory structure for Laravel[3] 4 is:

```
repo/
+-- app/
|   +-- assets/         # Source files
|       +-- main.css/
|       +-- main.js/
+-- bower_components/   # Bower packages
+-- public/
|   +-- assets/         # Build files
+-- awe.yaml            # Awe config
```

With the following configuration:

```
ASSETS:

    default:
        src:          app/assets/
        dest:         public/assets/
        bower:        bower_components/
        autoprefixer: true
```

[2] http://laravel.com/
[3] http://laravel.com/

CHAPTER 6

Cache files

Awe will create a hidden directory named `.awe` inside each project, which is used to hold cache files. This directory will automatically be ignored by Git (Awe will create a `.awe/.gitignore` file), but you may want to configure your editor to hide it.

6.1 Hiding in Sublime Text

In Sublime Text, go to Project > Edit Project and add a `folder_exclude_patterns` section:

```
{
    "folders":
    [
        {
            "path": ".",
            "folder_exclude_patterns":
            [
                ".awe"
            ]
        }
    ]
}
```

CHAPTER 7

Upgrading a project

Only breaking changes are listed here. For a full changelog please see the commits list[1] on GitHub (or run `git log`).

7.1 v0.1.0 (16 Nov 2014)

New config file format – see Project configuration files (page 7).

[1] https://github.com/alberon/awe/commits/master

CHAPTER 8

Quick reference

8.1 Command-line interface (`awe`)

8.1.1 Global commands

These commands can be run from any directory:

```
# Create an awe.yaml file in the current directory
$ awe init

# Display help
$ awe help

# Display the current version number
$ awe version
```

8.1.2 Project commands

These commands can only be run from a directory containing an `awe.yaml` config file (or any subdirectory):

```
# Build once
$ awe build
$ awe b

# Build then wait for further changes
$ awe watch
$ awe w
```

8.2 Configuration file (`awe.yaml`)

```
# Awe config - see http://awe.alberon.co.uk/ for documentation

ASSETS:

    groupname:                          # required (a-z, 0-9 only)
        src:          path/to/src/      # required
        dest:         path/to/build/    # required
        bower:        bower_components/ # optional (default: false)
```

```
        autoprefixer: true              # optional (default: false)

    anothergroup:                       # optional
        # ...
```

8.3 Assets directory structure

```
SOURCE                      DESTINATION                 NOTES
--------------------------  --------------------------  ---------------------------------------
src/                        build/
|                           |
|                           +-- _DO_NOT_EDIT.txt        Warning file (automatically generated)
|                           |
|                           +-- _bower/                 Symlink to bower_components/ directory
|                           |
|                           +-- _generated/             Compass-generated files (e.g. sprites)
|                           |   +-- nav-s71af1c74.png
|                           |
+-- _partials/              |                           Ignored (starts with _)
|   +-- reset.scss          |
|                           |
+-- _sprites/               |                           Compass sprite source images
|   +-- nav/                |
|       +-- edit.png        |
|       +-- save.png        |
|                           |
+-- _vars.scss              |                           Ignored (starts with _)
|                           |
+-- combined.css/           +-- combined.css            Combined (ends with .css)
|   +-- 1.css               |
|   +-- 2.scss              |
|   +-- 3-subdirectory/     |
|       +-- A.css           |
|       +-- B.scss          |
|                           |
+-- combined.js/            +-- combined.js             Combined (ends with .js)
|   +-- 1.js                |
|   +-- 2.coffee            |
|   +-- 3-subdirectory/     |
|       +-- A.js            |
|       +-- B.coffee        |
|                           |
+-- img/                    +-- img/                    Images are copied unaltered
|   +-- logo.png            |   +-- logo.png
|                           |
+-- sample1.css             +-- sample1.css             CSS file is copied
+-- sample2.scss            +-- sample2.css             Sass file is compiled
+-- sample3.js              +-- sample3.js              JavaScript file is copied
+-- sample4.coffee          +-- sample4.js              CoffeeScript file is compiled
|                           |
+-- subdirectory/           +-- subdirectory/           Directory structure is preserved
|   +-- A.css               |   +-- A.css
|   +-- B.scss              |   +-- B.css
|   +-- C.js                |   +-- C.js
|   +-- D.coffee            |   +-- D.js
|                           |
```

```
+-- vendor.css.yaml        +-- vendor.css              YAML import file (.css.yaml)
+-- vendor.js.yaml         +-- vendor.js               YAML import file (.js.yaml)
```

Note: It will also generate source maps – e.g. `combined.css.map` – but these are not shown for simplicity.

8.4 YAML import files

```
- _vendor/jquery.js          # Relative path to partial
- ../vendor/jquery.js        # Relative path to outside directory
- bower: jquery/jquery.js    # File inside bower_components/
```

CHAPTER 9

Design decisions

9.1 Introduction

A lot of time and effort has gone into making Awe, including a lot of back-and-forth about the best way to build it. Here I document some of the design decisions, both as a reminder for myself and to explain the thinking behind it to others.

9.2 Specific- not general-purpose

Awe was created to make it easier for the team at Alberon[1], a web/software development agency, to manage many different websites and web apps. Unlike Grunt[2], Gulp[3] and others, it is not designed to be a general-purpose task runner or build tool, but to perform specific tasks well.

9.3 System-wide installation

Before building Awe I tried using Grunt[4]. This required me to install Grunt and all the plugins I was using in each project, upgrade each project separately when new versions were released, and keep the Gruntfiles in sync so every project had the latest features & fixes as I added them. This was tedious enough with 3 projects – if it were expanded to all 25+ ongoing projects it would be a nightmare. So Awe is designed to be installed (and upgraded) only once, system-wide.

Future Plans

CoffeeScript, Compass, etc. are also installed system-wide, so every project must use the same version. In the future this could be changed to allow specific versions to be required for each project, and Awe would install/upgrade them for each project automatically (in the `.awe/` hidden directory). However, I would need to be convinced that this was better than just upgrading all projects at once – e.g. a non-backwards compatible change.

[1] http://www.alberon.co.uk
[2] http://gruntjs.com/
[3] http://gulpjs.com/
[4] http://gruntjs.com/

9.4 Unit tests

Because Awe is installed system-wide, backwards compatibility is especially important. So we have plenty of unit tests to ensure nothing breaks.

Note: "Backwards compatible" doesn't mean completely identical build output – for example, adding source maps meant adding extra comments to the build files, but they are still backwards compatible.

9.5 Conservative defaults

Future-proofing is also important for the same reason, so the default settings are quite conservative – features must be explicitly enabled in the config file, even if they are strongly recommended (e.g. Autoprefixer). Only features that are not expected to cause any problems (e.g. source maps) are enabled by default.

9.6 Minimal configuration

To ensure consistency between sites, only a minimum amount of configuration is allowed. It is limited to:

- Choosing the functionality to use (see above), and
- Allowing for necessary differences between projects (e.g. assets are different directories depending on the framework/CMS used)

In particular, config options should not be added to avoid making a decision[5] about the best solution.

9.7 YAML configuration

Many systems allow configuration files to be written in code (e.g. *Gruntfile.js*). While this allows more advanced customisation, I wanted to ensure consistency between sites and keep the configuration simple, which means limiting the options available.

If any extra functionality is required, it should be added to Awe itself, not added through custom project-specific code. This ensures it can be reused in other projects.

Future Plans

I may add hooks that can be called at certain points (on build, on deploy, etc.) when custom functionality is truely needed. These would most likely be external scripts (which can be written in any language) rather than Node.js functions.

9.8 Automatic mapping of asset files

There are no configuration options for *how* assets are built – the idea is anyone should be able to look at the source files and work out what the resulting build files will look like. This is especially important when working with frontend (HTML/CSS) developers who are not programmers, work on a lot of different projects and just want it to work.

[5] https://gettingreal.37signals.com/ch06_Avoid_Preferences.php

9.9 YAML import files

In an early alpha version of Awe, I used symlinks and *combined directories* (page 16) to merge vendor files with custom files. However, when viewing the directory over the network using Samba it was impossible to see which files were symlinks, therefore impossible to tell which files were custom and which were external (e.g. Bower packages). So symlink support was removed in favour of *YAML import files* (page 17).

9.10 No shorthand syntax in import files

In the *YAML import files* (page 17) you must always use a list, even if there is only one entry:

```
- ../vendor/jquery.js
```

You cannot shorten it to:

```
../vendor/jquery.js
```

This is to avoid confusing the user when they try to add a second entry to the file.

9.11 Limited file type support

Awe doesn't support the shorthand Sass[6] syntax (`.sass` files), Less[7] or several other languages purely because we (Alberon) don't currently use them. If we do decide to use them, we can add support for them in the future.

Future Plans

I would consider switching to a plugin-based architecture, more like Grunt, as long as Awe installed and upgraded them automatically in response to config options – i.e. it would not require the user to run `npm install` manually – and it didn't require any complicated configuration (unlike Grunt & Gulp).

9.12 Open source

Although Awe has a limited target audience, it is open source to allow other people to use it – particularly if a third-party takes over maintenance of a site/app we built. If anyone else wants to use it or improve it, that's absolutely fine. (Please do share your changes[8]!)

It also allows us to enjoy the benefits of open source – free hosting on GitHub[9], npm[10] and Read the Docs[11].

9.13 Flag deprecated features

Future Plans

[6] http://sass-lang.com/
[7] http://lesscss.org/
[8] https://github.com/alberon/awe/pulls
[9] https://github.com/alberon/awe
[10] https://www.npmjs.org/package/awe
[11] https://readthedocs.org/projects/awe/

If any features are deprecated in the future, Awe should warn the user whenever they are used *and* suggest an alternative. There should be no way to disable these warnings. This will ensure that most projects are upgraded early, so they do not break if that feature is eventually removed.

9.14 Runs in a terminal (SSH)...

9.14.1 ... not locally on Windows

Most of us at Alberon develop on Windows but use a Linux development server, editing files over a Samba network drive. This means a local GUI application would not be able to watch for file changes efficiently (e.g. see Prepros[12]).

9.14.2 ... not through a web server

Another option was to have it run automatically through the web server, rebuilding the files whenever they were requested – similar to Rails' asset pipeline[13]. This would have the advantage that it wouldn't be necessary to run Awe over SSH (which easy to forget if you're not used to it). However:

- It's more difficult to display errors this way (especially in CSS files)
- There's not always a 1-to-1 mapping of source to build files, making efficient compilation difficult
- It's slower to detect changed files, as they must be searched for each file loaded
- It would require more setup for each site

9.14.3 ... not in a browser (web app)

Another option would be to build an application frontend that runs in the browser and communicates with a server process using WebSockets. This would be a more friendly interface for less technical frontend developers, but require significant extra work to implement.

None of these are three options are impossible, but the industry seems to be moving toward command-line build tools anyway, so that seemed like the best solution for now.

9.15 Both asset building *and* deployment

Future Plans

Deployment is not yet available, but is planned for a future release.

I considered splitting asset building and deployment into two separate applications, so they could be installed independently, but:

- Awe is not meant to be a general-purpose build tool that many people use, so the benefits would be limited
- It's easier for me to maintain a single application than several smaller ones
- Combining them will make it easier to minify/compress assets as part of the deploy process

[12] https://github.com/subash/Prepros/issues/398#issuecomment-60480027
[13] http://guides.rubyonrails.org/asset_pipeline.html

CHAPTER 10

Contributing

10.1 Introduction

To submit a simple documentation change, simply edit the appropriate file on GitHub[1]. (There's even an Edit link in the top-right corner of each page!)

Warning: Not all markup is supported by GitHub – e.g. `:ref:` and `:doc:` – so the preview may not be exactly what appears in the online documentation. Don't let that put you off making changes, but if you're making substantial changes it would be better to clone the repository and *test it offline* (page 38) first.

If you want to submit a bug fix, the information below should help you to get started. Push your changes to a new branch on GitHub, then open a pull request[2].

If you want to suggest a new feature, I recommend opening an issue[3] to discuss the idea first, to make sure it will be accepted. (Or you can go ahead and develop it first if you prefer!)

10.2 System requirements

First make sure your system meets the main *system requirements* (page 3).

In addition, you will need:

10.2.1 Grunt (CLI)

To check if it's installed:

```
$ grunt --version
```

To install it:

```
$ sudo npm install -g grunt-cli
```

[1] https://github.com/alberon/awe/tree/master/docs
[2] https://github.com/alberon/awe/pulls
[3] https://github.com/alberon/awe/issues

10.2.2 Python & Sphinx

The documentation is generated by Sphinx[4], which is written in Python[5] and installed with pip[6].

To check if they're installed, run:

```
$ python --version
$ pip --version
$ sphinx-build --version
```

Installing Python & pip (on Debian)

```
$ sudo apt-get install python-pip
```

Installing Sphinx

```
$ sudo pip install sphinx sphinx-autobuild sphinx_rtd_theme
```

10.2.3 LaTeX (optional)

To build the PDF documentation, you will also need LaTeX installed. To check:

```
pdflatex --version
```

Installing LaTeX (on Debian)

```
$ sudo apt-get install texlive texlive-latex-extra
```

10.3 Installing Awe from Git

10.3.1 Download source code

Obtain a copy of the Awe source code, if you haven't already. If you are planning to make changes, it is best to fork the Awe repository on GitHub[7] first – then use your own username in place of `alberon` below.

You can install Awe into any location, but `~/awe/` would be a logical choice and is used below.

```
$ cd
$ git clone git@github.com:alberon/awe.git
```

[4] http://sphinx-doc.org/
[5] https://www.python.org/
[6] https://pypi.python.org/pypi/pip
[7] https://github.com/alberon/awe/fork

10.3.2 Install dependencies

```
$ cd awe
$ npm install
```

This will:

- Install Node.js dependencies using npm
- Install Ruby dependencies using Bundler
- Compile the source files (from IcedCoffeeScript[8] to JavaScript)
- Run the test suite (using Mocha[9])

At this point it should be possible to run Awe by specifying the path to the executable:

```
$ ~/awe/bin/awe --version
```

10.3.3 Make it the default version (optional)

If you would like to run `awe` directly, instead of using the full path, run:

```
$ export PATH="$HOME/awe/bin:$PATH"
```

This will only last until you close the terminal session.

10.3.4 Upgrading Awe from Git

```
$ cd ~/awe
$ git pull
$ npm install
```

10.3.5 Uninstalling

Simply delete the source directory:

```
$ cd
$ rm -rf awe
```

10.4 Source code

The source code is in `lib/`. It is written in IcedCoffeeScript[10] – and you will need to understand `defer` and `await` as they are used extensively.

To compile it, run:

```
$ grunt build-lib
```

Alternatively, to compile everything at once (source code, documentation and man pages – excludes PDF docs):

[8] http://maxtaco.github.io/coffee-script/
[9] http://visionmedia.github.io/mocha/
[10] http://maxtaco.github.io/coffee-script/

```
$ grunt build
```

Or to build everything at once and then watch for further changes and rebuild automatically (**the recommended method**):

```
$ grunt watch
```

In each case the compiled JavaScript code is written to `lib-build/`, and you can run the `bin/awe` executable script to run it.

10.5 Unit tests

Please ensure that every important function and bug fix has corresponding unit tests, to ensure backwards compatibility.

The unit tests are in `test/`. They are written in regular CoffeeScript[11].

To run them all:

```
$ grunt test
```

To run a single test suite, add the filename without the extension:

```
$ grunt test:AssetGroup  # -> test/AssetGroup.coffee
```

When you run `grunt watch`, it will:

- Automatically run any test suite that is modified
- Run the appropriate test suite when any file in `lib/` is modified (e.g. when `lib/AssetGroup.iced` is modified, `test/AssetGroup.coffee` will be run)

You should manually run `grunt test` before committing your changes, to ensure that *all* tests are still passing.

10.6 Documentation

Documentation is in `docs/`. It is written in reStructuredText[12] and converted to HTML and PDF formats by Sphinx[13].

To build the HTML docs:

```
$ grunt build-docs-html
```

When you run `grunt watch`, it will automatically rebuild whenever a file in `docs/` is modified.

Warning: When using `grunt watch`, Sphinx will only rebuild modified files. When one file references another (e.g. the table of contents), some information may be out of date. To force it to rebuild all files, run `grunt docs` manually.

10.6.1 PDF documentation

The PDF documentation takes several seconds to generate, so it is not built automatically. To build the PDF docs:

[11] http://www.coffeescript.org/
[12] http://docutils.sourceforge.net/rst.html
[13] http://sphinx-doc.org/

```
$ grunt build-docs-pdf
```

10.6.2 Sphinx markup reference

I found the following documents useful when writing the documentation:

- reStructuredText quick reference[14]
- Admonitions list[15] (`note::`, `warning::`, etc.)
- Code examples markups[16] (`code-block::`, `highlight::`)
- Other paragraph-level markup[17] (`versionadded::`, `deprecated::`, etc.)
- Inline markup[18] (`:ref:`, `:doc:`, etc.)
- Table of contents[19] (`toctree::`)

10.6.3 Heading styles

The following code styles are used for headings:

```
################################################################################
 Page title (80 hashes)
################################################################################

================================================================================
 Section title (80 equals signs)
================================================================================

----------------------------------------
 Heading 2 (40 hypens)
----------------------------------------

Heading 3 (full stops)
......................
```

10.6.4 Custom admonitions

I found it necessary to make some custom admonitions (alert boxes) using HTML classes that are available in the Read the Docs theme[20]:

```
.. admonition:: Alberon Note
    :class: note wy-alert-success

    This is a note for staff at Alberon specifically...
```

[14] http://docutils.sourceforge.net/docs/user/rst/quickref.html
[15] http://docutils.sourceforge.net/docs/ref/rst/directives.html#admonitions
[16] http://sphinx-doc.org/markup/code.html
[17] http://sphinx-doc.org/markup/para.html
[18] http://sphinx-doc.org/markup/inline.html
[19] http://sphinx-doc.org/markup/toctree.html
[20] https://github.com/snide/sphinx_rtd_theme

```
.. admonition:: Future Plans
    :class: note

    This is something I plan to add in the future...
```

For other classes see the Wyrm documentation[21].

10.7 Updating dependencies

Before updating any dependencies, remember to check the changelogs to ensure they are compatible.

10.7.1 Node.js

To check for updates:

```
$ npm outdated
```

To install updates:

```
$ npm update
```

(You will need to update the version number in `package.json` first to install some updates.)

10.7.2 Ruby

To check for updates:

```
$ bundle outdated
```

To update the Ruby gems to the latest version:

```
$ grunt update-gems
```

This will install the latest versions and update *Gemfile.lock*.

10.8 Releasing a new version

- Check the documentation is up-to-date
- Update Upgrading a project (page 25) (if necessary)
- Run `grunt deploy`

[21] http://wyrmsass.org/section-2.html