Aldryn Boilerplate Standard Documentation

Release 3.3.0

Divio AG

April 03, 2017

Contents

Aldryn Boilerplate Bootstrap 3 is the most complete **django CMS** based Boilerplate for rapid development. It uses the full potential of the Bootstrap framework for developing responsive, mobile-first projects on the web, and implements various best practices from within the front-end community.

This Boilerplate can be used with standalone django CMS websites as well as on the Aldryn cloud platform.

The latest stable version is available on GitHub - https://github.com/aldryn/aldryn-boilerplate-bootstrap3.

CHAPTER 1

Documentation

General

Note: Welcome to the starting point of this documentation. Throughout each section we familiarise you with the Guidelines, Structure, Testing and other essential modules for our front-end code. The goal is to create a common coding ground for all developers.

What's inside

Note: This Boilerplate includes and configures a number of components.

Sass

For CSS pre-processing, we use Sass. In particular, we use:

- LibSass for fast SASS compilation
- Gulp JS and the gulp-sass plugin to compile SASS files
- the official Sass port of Bootstrap

All styles should be created in `/private/sass`, and will be compiled to `/static/css`.

Bootstrap

The full Bootstrap library is imported via the Sass component and the JavaScript component.

Note: Aldryn Bootstrap 3 uses a 24 column based grid setting instead of the default 12. You can change this setting in `private/sass/settings/_bootstrap.scss`.

The Glyhpicon icon set has been disabled in favour of the Font Awesome icon set.

Font Awesome

The Font Awesome library offers a larger and better variety of icons than the Bootstrap defaults. Additional utility classes are also available.

The library is similarly integrated as bootstrap-sass within the `libs` folder.

JavaScript

We are implementing the latest **2.x.x** versions of jQuery as they are released. In addition we encourage the use of class.js, a simple library that helps out with the modular pattern in JavaScript.

- http://jquery.com
- https://github.com/FinalAngel/classjs

In addition several commonly-used shims are available to you including:

- The HTML5 Shiv
- Respond.js
- <swfobject>
- Outdated Browser
- console.log wrapper

Addons

We are currently implementing the select2.js bootstrap version as default addon.

Gulp

We use Gulp to manage our frontend workflow.

Template Language

As this is a django CMS based boilerplate, naturally we are using the Django template language.

In order to implements assets efficiently, django-sekizai and aldryn-snake are implemented within the `base_root.html` template. This gives you the `{% addtoblock "js" %}{% endaddtoblock %}` and `{% addtoblock "css" %}{% endaddtoblock %}` template tags in addition to the django defaults.

Example

```
{% load sekizai_tags %}
{% addtoblock "css" %}<link href="{% static 'css/theme.css' %}" rel="stylesheet">{% endaddtoblock %}
{% addtoblock "js" %}<script src="{% static 'libs/jquery.min.js' %}"></script>{% endaddtoblock %}
```

- http://docs.django-cms.org

Configuration

There are several **configuration files** included such as:

- EditorConfig within `.editorconfig`
- CSSComb within `.csscomb.json`

- ESLint within `.eslintrc.json`
- SCSS-Lint within `scss-lint.json`

Please mind that they are ignored if your editor doesn't support them.

Installation

Note: The following dependencies should be installed on your system in order to work with this Boilerplate.

- Sass: http://sass-lang.com/
- Bootstrap: https://github.com/twbs/bootstrap-sass
- Node JS: http://nodejs.org/
- Gulp: http://gulpjs.com/

You can find most installation steps within osx-bootstrap but in short:

1. run `brew install node` when using Homebrew
2. run `curl -L https://npmjs.org/install.sh | sh`
3. run `npm install -g bower`
4. run `npm install -g gulp`

At last make sure you correctly configured your paths.

Setup

Run the following commands to install all requirements from within the root of the package:

- `npm install` to install the requirements from `package.json`
- `bower install` to install the requirements from `bower.json` via `.bowerrc`

Gulp Commands

Warning: Please mind that `gulp browser` starts *browserSync* which tries to connect to a server. A Django server can be started from within `tools/server`. Refer the General section for additional information.

All front-end related tasks are handled via the Gulp task runner:

- `gulp` runs the gulp defaults
- `gulp browser` connects to a given server (django) and runs live reload on a separate IP
- `gulp lint` starts all linting services using `.eslintrc.json` and `scss-lint.json`
- `gulp preprocess` optimises images within `/static/img` and compiles YUIDoc into `static/docs`
- `gulp sass` to compile the stylesheets
- `gulp tests` runs the test suite
- `gulp watch` runs the gulp watch defaults

We also offer some standalone commands:

- `gulp bower` to install the bower dependencies
- `gulp images` optimises images within `/static/img`
- `gulp icons` to create a custom icon webfont
- `gulp docs` compiles YUIDoc into `static/docs`
- `gulp lint:javascript` runs JavaScript linting
- `gulp lint:sass` runs Sass linting
- `gulp tests:unit` runs unit tests
- `gulp tests:integration` runs integration tests
- `gulp tests:watch` runs tests in debugging mode

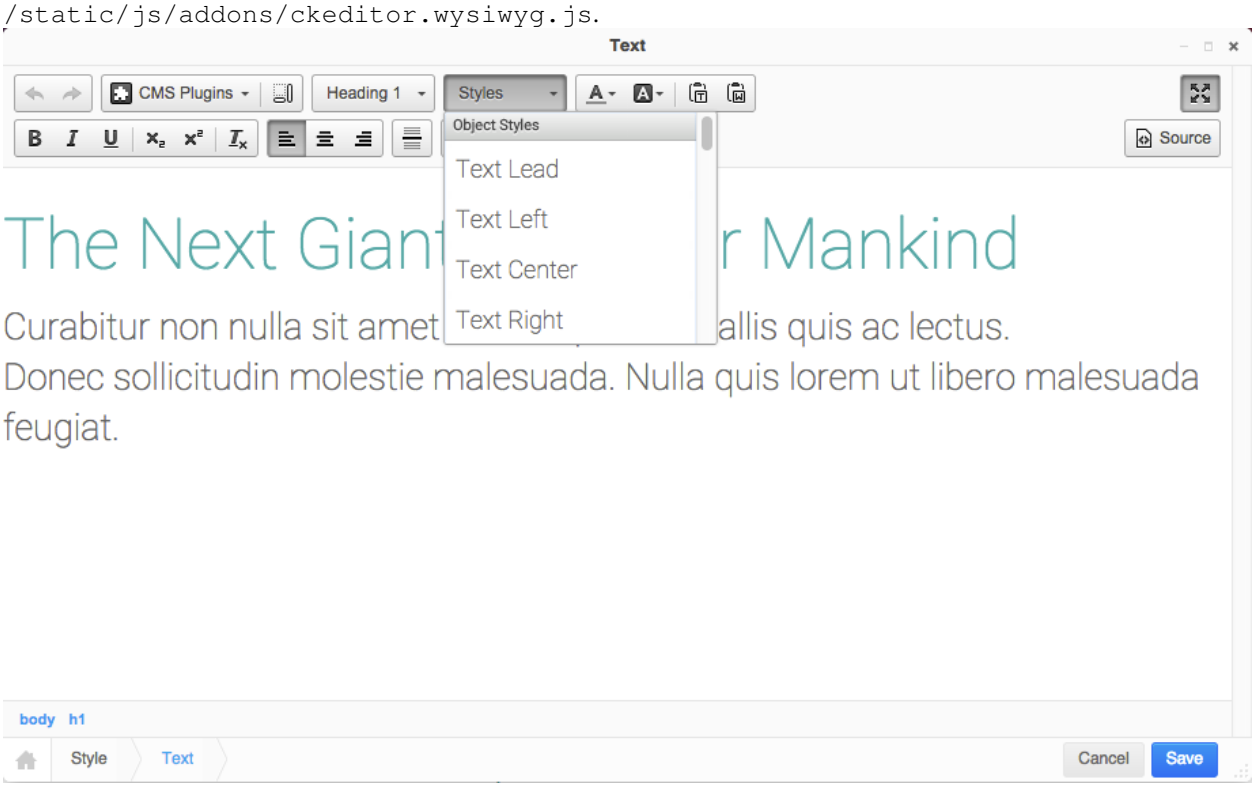
We love code over configuration.

Configuration

Note: The Boilerplate ships pre-configured and runs out of the box if the Installation steps are followed properly. However most components can be freely configured.

WYSIWYG

The CMS allows for custom style sets within the editor. This ables the user to choose certain presets or colours. We already added the general Bootstrap utilities for you. The file can be found at: `/static/js/addons/ckeditor.wysiwyg.js`.



Custom Icons

We added support for custom icon-font generation through Gulp. There are some configuration steps required if you want to use them:

1. Add your SVG fonts to `/private/icons`. Gulp gets all SVG files from the `/private/icons/**/*.svg` pattern and generates the fonts for you.
2. Run `gulp icons` to generate the icon-font
3. Uncomment `// @import iconography;` from `/private/sass/layout/_all.scss` to include it in your gulp build

The `gulp icons` command will automatically generate the `/private/sass/layout/_iconography.scss` file where you find the class reference and mixins for all icons.

The generated icon-font will use the `.icon` css namespace for all custom icons. We recommend using the `icon(*)` mixin instead of `@extend .icon-*`.

Basic usage

Once installed in an Aldryn or django CMS project, Aldryn Boilerplate Bootstrap 3 is ready to use.

In your project

See Templates for guidelines on how to set up your project templates so that they take advantage of what it has to offer.

Fundamentally, if your project's templates inherit from the base.html template, they'll be furnished with the classes, elements, hooks and other things they need.

In your applications

Your applications, if they are aware of Aldryn Boilerplate Bootstrap 3, can also take advantage of it.

You could simply make your application assume that Aldryn Boilerplate Bootstrap 3 will be available. That's not ideal though, because it will be off-putting to people who don't want to have to use it. A reusable application should have requirements that are as generic as possible, not based on a particular frontend framework.

So, although it means a little more work for you, you should **also** provide more generic frontend (templates, CSS etc) support for the application, and if you like, for other Boilerplates too.

At the very least, the developers who use your application will find it easier to create templates and static file for it that support their own frontend conventions if they can start with simple ones.

Aldryn Boilerplates

To make this easier, use the Aldryn Boilerplates application.

This provides support for multiple Boilerplates, allowing you to offer rich frontend machinery compatible with Aldryn Boilerplate Bootstrap3 for those who want it, and generic frontend files for those who don't, in a way that the correct set will automatically be chosen.

You can also add support for *other* Boilerplates, by adding the frontend files to namespaced directories in your application. This example of an application named *aldryn_addon* mentions only templates for sake of simplicity, but the same principle applies to static files:

```
aldryn_addon
- templates/  # the generic templates
|  - aldryn_addon/
|  - base.html
- boilerplates/  # templates for particular Boilerplates
|  - aldryn_boilerplate_bootstrap3/
|     - templates/
|        - aldryn_addon/
|        - base.html
- some_other_boilerplate/
   - templates/
      - aldryn_addon/
      - base.html
```

See Aldryn Boilerplates for more.

Guidelines

Note: This section describes guidelines for several front-end related technologies. We advise you to follow them. However it is sometimes necessary to break the rules in which case you have to watch this video and **leave a comment** describing why this was necessary.

General

Note: There are global guidelines which affect every single language, file or folder.

Standards

Important:

- **Validate** your code through the W3C validators.
- There is something called Accessibility.
- Don't forget about **HiDPI**, **Retina** and **High Resolutions** displays.
- Proper fallbacks should be available if a connection is slow or features are disabled.
- Progressive enhancement, graceful degradation and responsive design are buzzwords you care about.
- Develop with modularity and extensibility in mind.
- Documentation is your friend.

Spacing

Important:

- Use **4 spaces** for indentation.

Not 2, 3 or 8 – no tabs – if you are able to do 3 $^{3/4}$, that's good enough

Line Length

Important:

- Don't breach **120 characters** per line.

Not even for HTML. We even encourage you to use 80 characters per line. Yes, screens have got much bigger over the last few years, but your brain hasn't. Better to use screen estate for splits, anyway.

Naming

Important:

- **lowercase**, **camelCase** or **hyphened separation** are all good; use **no special characters** except for underscore _.
- Use dashes - for file naming, unless expressly counterindicated (e.g. in HTML template names).
- Always use full words instead of abbreviations: `number` is better than `nr`.
- BEM is a nice methodology to be aware of.

"There are only two hard things in Computer Science: cache invalidation and naming things" – Phil Karlton

Quotes

Important:

- We always use **double `"."`** quotes for everything, *except in JavaScript*, where we use **single `'.'`** quotes.

Comments

Note: If peppering your code with lots of comments is good, then having zillions of comments in your code must be great, right? Not quite. It doesn't make sense to comment every step your code makes, or to comment on things that don't need to be explained.

Comments in code should describe:

- **what** is being done
- **why** it's being done

They do not need to describe:

- **how** it is being done (the code already shows this)
- what you are thinking about

Section Comments

In addition to the regular comments, we introduced the *section comment*. Use this style to separate large chungs of logic (which you should generally avoid). The line is exactly 80 characters long:

```
// ###############################################################################
// NAME
```

Inline Comments

When using comments inline, make use of the appropriate formats:

- `{# ... #}` or `{% comment %} ... {% endcomment %}` for Django templates and **never** `<!-- ... -->`
- `// ...` and `/* ... */` for Sass and JavaScript

Notes

We also use several comment helpers which, if configured in your editor, add additional highlighting to your code:

FIXME:
to annotate problems with the code

```
function Calculator() {
    // FIXME: shouldn't use a global here
    total = 0;
    ...
}
```

TODO:
to annotate solutions to problems with the code

```
function Calculator() {
    // TODO: total should be configurable by an options param
    this.total = 0;
    ...
}
```

DOCS:
provides a simple docs link

```
// DOCS: https://django-cms.readthedocs.org/en/latest/
```

Formatting

Comments

- Add proper whitespace.
- In general use lowercases except for the *Notes*.

```
bad
//TODO: THIS NEEDS ADDITIONAL REVIEW
//
// square root of n with Newton-Raphson approximation
/**
 * Contains various helpers, feel free to extend and adapt
 */
```

```
good
// TODO: this needs additional review
// square root of n with Newton-Raphson approximation
/**
 * Contains various helpers, feel free to extend and adapt
 *
 * @class Utils
 * @namespace Cl
 */
```

YUIDoc

In 3.3.0 we introduced YUIDoc which uses syntax similar to JSDoc in order to further improve JavaScript documentation. We encourage using this style within your code, as shown in `/static/js/addons/cl.utils.js`.

Markup

Note: In addition to the General guidelines, the following sections describe markup specific rules.

Naming

Important:

- Use **underscores** for HTML file naming.
- Use lowercase for **all** attributes.

```
// bad
two_column-template.html, tpl-master.html, askForAdditionalInformation.html

<DIV class="box boxHighlighted" DATA-rel="#my_modal"> ... </DIV>
```

```
// good
two_column_template.html, tpl_master.html or ask_for_additional_information.html

<div class="box box-highlighted" data-rel="#my-modal"> ... </div>
```

Indentation

Important:

- Always add an indent after Django tags such as `{% if %}`, `{% forloop %}`, `{% block %}` and so on.
- Use single lines within `{% addtoblock %}` for **files** and multilines for `<code>`. It is important because of how sekizai works. Basically if two scripts are added through `addtoblock` and the contents of the block are the same they are merged. That way you never have duplicate jQuery's on the page. The caveat than is that the whitespace around that script tag must match. To avoid mistakes we always do them in single line.
- **Code readability** always wins.

```
// bad
{% block content %}
<div class="plugin-blog">{% if true %}<p>Hello World</p>{% endif %}</div>
{% endblock content %}

{% addtoblock "js" %}
<script src="{% static "js/libs/jquery.min.js" %}"></script>
{% endaddtoblock %}
{% addtoblock "js" %}
    <script>
        jQuery(document).ready(function ($) {
            alert('hello world');
        });
    </script>
{% endaddtoblock %}
```

```
// good
{% block content %}
    <div class="plugin-blog">
        {% if true %}
            <p>Hello World</p>
        {% endif %}
    </div>
{% endblock content %}

{% addtoblock "js" %}<script src="{% static "js/libs/jquery.min.js" %}"></script>{% endaddtoblock %}
{% addtoblock "js" %}
<script>
jQuery(document).ready(function ($) {
    alert('hello world');
});
</script>
{% endaddtoblock %}
```

IDs vs Classes

Important:

- Avoid IDs wherever possible.
- Where it's necessary to use IDs, always use **unique names**.

You should **always** use classes instead of IDs where you can. Classes represent a more OOP approach to adding and removing style sets like `box box-wide box-hint`.

Try to avoid declaring ID's at all. They should only be used to reference form elements or for in-page navigation in which case you need to make the name **absolutely unique**.

```
// bad
<div class="box box-highlighted" id="box-8723"> ... </div>
<!-- IDs only for navigation jumper through <a href="#page-anchor-team"></a> -->
<div id="team"></div>
<!-- IDs only for form elements -->
<label for="firstname">Name</label>
<input type="text" name="firstname" id="firstname">
```

```
// good
<div class="box box-highlighted box-8723"> ... </div>
<!-- IDs only for navigation jumper through <a href="#page-anchor-team"></a> -->
<div id="page-anchor-team"></div>
<!-- IDs only for form elements -->
<label for="field-id12-firstname">Name</label>
<input type="text" name="firstname" id="field-id12-firstname">
```

Modularity

Important: Try to keep HTML structure simple, avoiding unnecessary elements. It is sometimes easier to use a single div with a single class rather than multiple divs with multiple classes.

For example, lets take a look at the following code snippet:

```
<div class="addon-blog">
    <h2>My Blog</h2>
    <p>Hello World</p>
</div>
```

We should build modular HTML, and take pains to avoid type selectors. Add additional classes for lead, content, author, meta info, tags and so on. The content section itself can then contain the usual HTML code:

```
<div class="addon-blog">
    <h2 class="blog-heading">My Blog</h2>
    <p class="blog-lead">Hello World</p>
    <div class="blog-content">
        <h3>Details</h3>
        <p>More</p>
        <p>Content</p>
    </div>
    <div class="blog-author">Dummy Man</div>
    <ul class="blog-tags tags">
        <li class="blog-tag-items"><a href="#">News</a>
        <li class="blog-tag-items"><a href="#">Blog</a>
        <li class="blog-tag-items"><a href="#">Tags</a>
    </ul>
</div>
```

Styles

Note: In addition to the General guidelines, the following sections describe stylesheet-specific rules.

Naming

Important:

- Use **lowercase** in SCSS file names.
- Use only **dashes** in class/ID names.

```
// bad
Search.scss, marketingSite.scss or theme-dark-blog.scss

class="blog blogItem blog_item__featured"
```

```
// good
search.scss, marketing_site.scss or theme_dark_blog.scss

class="blog blog-item blog-item-featured"
```

Nesting

Important:

- Don't overuse nesting; nest elements to a maximum of **4 indents**.

With great power comes great responsibility (just wanted to throw that in here). When writing in **Sass** or **Less** laziness can have performance implications. While nesting is very powerful, we should avoid unnecessary levels or blocks that can be simplified.

```
// bad
.nav-main {
    ul {
        @extend list-reset;
        li {
            padding: 5px 10px;
            a {
                color: red;
            }
        }
    }
}
```

```
// good
.nav-main {
    ul {
        @extend list-reset;
    }
    li {
        padding: 5px 10px;
    }
    a {
        color: red;
    }
}
```

Formatting

Important:

- Always add a space after the colon `:`.
- Only write one CSS property per line.
- Avoid using selectors such as `div.container` or `ul > li > a` (i.e. ad-hoc, non-namespaced) to determine specificity.
- Write colour values in lowercase and avoid colour names.

```
// bad
article.item {
    color: white;
    padding: 10px; margin-left: 0; margin-top: 0; margin-bottom: 10px;
    background-repeat: no-repeat;
    background-position: left top;
}
```

```
// good
.item {
    color: #fff;
    padding: 10px;
    margin: 0 0 10px 0;
    background: no-repeat left top;
}
```

Ordering

Important:

- Use block-style, and group elements below.
- See `scss-lint.json` for a comprehensive ordering example.

1. includes (mixins)
2. extending
3. visibility, position
4. color, font-size, line-height, font-* (font relevant data)
5. width, height, padding, margin (box model relevant date)
6. border, background (box style data)
7. media, print (media queries)
8. :after, :before, :active (pseudo elements)
9. nested elements or parent referencing selectors

Note: Combine attributes such as background-image, background-color, background-repeat into a single line `background: #fff url("image.png") no-repeat left top;` when it makes sense. But remember, that a shorthand like `background` cannot be overridden with just `background-image`, so use wisely!

Example

```
.addon-blog {
    // mixins
    @include border-radius(3px);
    @include box-shadow(0 0 2px #eee);
    // extending
    @extend .list-unstyled;
    // styles
    display: inline;
    position: relative;
    z-index: 1;
    color: white;
    font-size: 16px;
    line-height: 20px;
    width: 80%;
    height: 80%;
    padding: 5px;
    margin: 0 auto;
    border: 2px solid #ccc;
    background: #ddd;
    // desktop and up
    @media (min-width: $screen-md-min) {
        display: block;
    }
    // pseudo elements
    &:active,
    &:hover {
        color: black;
    }
}
```

JavaScript

Note: In addition to the General guidelines, the following sections describe JavaScript specific rules. Code Conventions for the JavaScript Programming Language should be your Bible.

Naming

Important:

- Use **dot** annotation `test.base.js` for JavaScript file naming.
- Use a library's prefix, such as `cl.xplorer.js` or `jquery.tablesorter.js`, for file naming.
- Name `variablesLikeThis`, `ClassesLikeThis`, `CONSTANTS_LIKE_THIS` and `events-like-this`.
- Use the `js-` prefix when working with JS-related selectors and do not add styling to it.
- Never use comma separation for variable declarations like `var a, b, c;`.
- Never use $ for variable names such as `var $el = $('.el');`.
- We are using the `Cl.` singleton for all custom JavaScript code.

When using jQuery to refer to a DOM instance, always use the `js-` prefix to separate styles from JavaScript functionality. For example: `<div class="addon addon-gallery js-addon-gallery"></div>`.

In this example, *addon* and *addon-gallery* have styles attached to them, *js-plugin-gallery* refers to the JavaScript functionality attached to the DOM element.

Even when removing the JS class (or just waiting for JavaScript to kick in), the addon should still look good.

```
// bad
CL.Utils.js, jquery_tooltip.js, testWebsiteCreateNew.js

var $jquery, current_state;
var test-website-create-new;
```

```
// good
cl.utils.js, jquery.tooltip.js or test.website.create.new.js

var jquery;
var currentState;
var nextIndexValue;
```

Formatting

Important:

- Always declare variables on top of the functions and not in between.
- Always use semicolons and full brackets, except in shorthand like `var i = (true) ? 'yes' : 'no';`.
- Use proper spaces for `if (true) {} else {}` or `function () {}`.

```
// bad
function(cont){
    var c = $(cont);
    if(c.length) {
        // do something
    }
    else
    {
        // so something else
    }
}
```

```
// good
function (container) {
    var container = $(container);
    if (container.length) {
        // do something
    } else {
        // so something else
    }
}
```

Implementation

Important:

- Keep `<script>` and the following starting closure on the same level.
- Separate all script tags using `{% addtoblock "js" %}{% endaddtoblock %}`.
- Never use JavaScript attributes on HTML elements such as `onclick=""` or `onload=""`.
- Don't add inline JavaScript within HTML, implement JavaScript through **files only**. Instantiate functionality from within the JavaScript file instead.

```
// bad
<div class="dashboard" id="dashboard"> ... </div>
{% addtoblock "js" %}
<script src="{% static "js/addons/cl.dashboard.js" %}"></script>
{% endaddtoblock %}
<!-- javascript gets initialised inside the template -->
{% addtoblock "js" %}
<script>
jQuery(document).ready(function () {

    Cl.dashboard.init('#dashboard');

});
</script>
{% endaddtoblock "js" %}
```

```
// good
<div class="dashboard js-dashboard" data-dashboard="..."> ... </div>
<!-- javascript gets initialised within the file -->
{% addtoblock "js" %}<script src="{% static "js/addons/cl.dashboard.js" %}"></script>{% endaddtoblock
```

Patterns

Important:

- Use the singleton pattern to avoid globals.
- Use the module pattern to structure code.
- Avoid the functional pattern

Structure

Note: This section covers naming conventions for folders and sub-directories. The correct placing of files is imperative for a common shared structure. It is easier to find code when you already know where it's going to be.

General

Note: Let's cover the core structure of this Boilerplate consisting of the main folders:

```
docs/
private/
static/
templates/
tests/
```

The starting point for each entry is always named "base"'", with the appropriate file extension. For HTML `base.html`, Sass `base.scss`, JavaScript `base.js` – you get the idea. This way you always know which file you should look after **first**. Lets take a closer look at each individual folder:

docs/

The full documentation is stored within `/docs` and is compiled into `/docs/_build` when running `make run`. The documentation is automatically pushed to Read the Docs once something is committed to the *master* branch. More information on how to contribute to the documentation can be found within the Contribution section.

private/

Important: This folder is **not published**, nor touched by preprocessing or other build libraries. Anything in here should be and remain safe.

This folder is intended for storing preprocessing library code (Sass, Less, Coffee, HAML, etc). Simply create a folder within `/private` with appropriate name: `/sass`, `/less` or `/haml` and so on as required. Always place required configuration files within the `/private` root.

```
private/
- sass/
|   - base.sass
- config.rb
```

Hint: We are using `/sass` as folder name and not `/scss` as the language itself is called Sass. Always use the full written acronym.

static/

Important: This folder is publicly available, all files can be accessed via `http://yourwebserver/static/`.

The default folder layout looks as follows:

```
static/
- css/
- fonts/
- img/
- js/
- swf/
- ...
```

If folders are not required, just simply remove them. When a folder reaches a certain file count, make use of grouping and create additional sub-directories such as `/static/img/icons` or `/static/js/addons/jquery`.

templates/

All django templates should be allocated within the `/templates` folder. This also applies for apps or inclusion files. When using Haml, set your configuration so templates get compiled into `/templates`.

The default *index.html* is always `/templates/base.html`.

Global inclusion files are placed within `/templates/includes`. Addons normally have their own */includes* folder so they are not overcrowding the structure.

tests/

The test suite is described in more depth within the Testing section.

Private

Note: Let's have a closer look at the **Sass** setup within `/private` and explain how we structured the code in there. These principles can be expanded to other preprocessors options such as **Less** or **HAML**.

Every folder within `/private/sass` has a file called **_all.scss**. This file ultimately gets imported by `/private/sass/base.scss` which gets compiled into `/static/css/base.css`. Update the *_all.scss* file to include additional modules. To keep the file simple, do not include files directly within *base.scss*.

Let's cover the folders individually:

addons/

If a component is plug-and-playable, it probably belongs in here. Good examples are jQuery plugins or Aldryn addons. Sometimes larger application such as a shop might also be pluggable. If this is **not** the case, they belong in the `/sass/sites` directory.

> **Warning:** You will always encounter the question whether to place a component within `/sass/addons` or `/sass/sites`. In case of doubts, use the **sites folder**.

layout/

We consider the general look and feel as the *layout* of a website or application. This might include the typography, header and footer, icons or the printable version. The layout can be broken down into further parts if a website gets very large. We advise in general against this strategy and rather prefer to use `/sass/sites` to create specific layouts and derive from a global common style guide.

> **Warning:** Everything that targets a specific element, such as custom styles for Bootstrap components or a specific form error, belongs in `/sass/addons` or `/sass/sites`.

libs/

All independent files are placed within this folder. This implies that the order of inclusion does not matter within `/sass/libs/_all.scss`.

Hint: Libraries are, in their very core, plug-and-playable. The main difference between libraries and other plug-and-

play components is, that if a library is removed, things will break.

mixins/

This folder is used to store additional functions or mixins which are not part of the default bootstrap eco-system.

We provide already some helper functions such as `em(12px, 16px)` that calculates the pixel values from a given size in relation to the parent size.

Additionally we have mixins for managing *z-index* layers and *hide-content*.

settings/

It is very useful to store values, that are used more than twice, within their own variable. We even encourage storing **all colour values** within the settings. **Don't repeat yourself**. Create a sub-structure, similar to `/sites` if the structure becomes more complex.

> **Warning:** Do **not** add additional variables to `/private/sass/settings/_bootstrap.scss`. This file is reserved for **Bootstrap-only** variables. Use `/private/sass/settings/_custom.scss` instead.

sites/

Besides */addons* you will work mostly within the `/private/sass/sites` folder. All custom elements that are in general not plug-and-playable, fixed into the website somewhere or specific components, get thrown in here.

This will force you to devise and adhere to structure patterns. Here are some examples depending on the requirements for your project:

Note: Multisite Setup

Let's assume you create one style guide sharing different marketing websites or applications - your structure might look something like:

```
sites/
- application/
|  - _all.scss
|  - _general.scss
|  - _wizard.scss
- marketing/
|  - _all.scss
|  - _layout.scss
|  - _addons.scss
- _application.scss (imports application/_all.scss)
- _marketing.scss (imports marketing/_all.scss)
```

Note: Theme Setup

If you are using different themes for the same markup, your structure might look something like:

```
sites/
- dark_theme/
|  - _all.scss
|  - _header.scss
```

```
|  - _footer.scss
- white_theme/
|  - _all.scss
|  - _header.scss
|  - _footer.scss
- dark_theme.scss (imports dark_theme/_all.scss)
- white_theme.scss (imports white_theme/_all.scss)
```

Static

Note: As `/static` is publicly accessible, avoid adding sensitive files into this directory.

Keep the **root path** of `/static` simple and clean. Only favicons should be placed there. They ultimately get picked up by the `base_root.html` template.

css/

CSS gets automatically compiled via `/private/config.rb` into this folder. You can add additional files such as `*.htc` if required. But **always add CSS files through Sass**.

fonts/

All fonts should be placed here including icon fonts. You can create sub-directories to create a better overview. This folder might not be required if you are implementing fonts via services such as Google Fonts or fonts.com.

img/

Demo images (which might be later integrated as media files via Filer) should be placed within `/static/img/dummy`. This folder will be ignored by the `gulp preprocess` and `gulp images` commands.

Make use of grouping and create additional sub-directories such as `/static/img/icons` or `/static/img/visuals` if the file count seems to be excessive.

js/

The same structure approach as described within Private is applied to the JS directory. `/layout`, `/settings` and `/sites` are not required, but may be used. jQuery is an essential part and should be treated the same as the Bootstrap component.

swf/

Old school, currently only required to use `/static/js/libs/swfobject.min.js`.

Templates

Note: Aldryn Boilerplate Bootstrap 3 follows django CMS good practices, and provides three layers of site template inheritance using `{% extends %}`. See Django template engine.

From the top down the three layers are:

- *user-selectable page templates* (`base.html`), which inherit from:
- `base_root.html`

base_root.html

`base_root.html` sets up the components that will rarely if ever need to be changed, and that you want to keep out of sight and out of mind as much as possible.

It contains fundamental HTML elements (`<html>` `<body>` and so on) so that these don't need to be managed in inheriting templates.

It is also intended to be almost wholly content-agnostic - it doesn't know or care about how your site's pages are going to be structured, and shouldn't need to. To this end it provides an empty `{% block extend_root %}{% endblock %}`, that inheriting templates will override to provide the page's content.

In addition, Addons such as Aldryn News & Blog in the Aldryn Collection family of applications are designed to use the same JavaScript frameworks throughout, so there is no need for references to them to be made anywhere else than `base_root.html`.

base.html

`base.html` is the template that *designers* will be most interested in. It fills in the bare HTML elements of `base_root.html`, and allows page content structures and layouts (headings, `divs`, navigation menus and so on) to be created within `{% block extend_root %}`.

`base.html` contains an *empty* `{% block content %}`, that - in templates that extend it - is filled with `{% placeholder content %}` as well as width cues for images etc.

User-selectable page templates

Finally, users can select templates that inherit from `base.html`. Even if your project has one 'standard' template and some minor variations, it is wise for *all* of them to inherit from a `base.html`, so that they can all be edited independently. Even if your 'standard' template changes nothing in `base.html`, you should not be tempted to make `base.html` selectable by the user.

The following templates are always required:

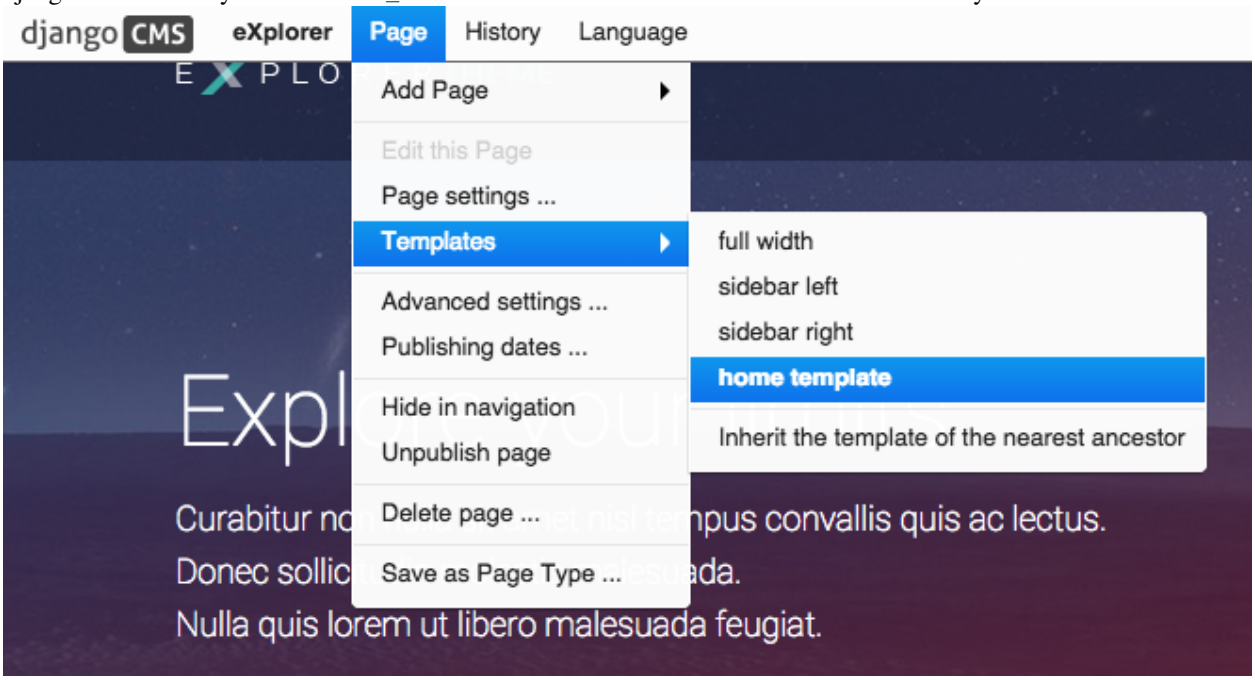
- `404.html` for 404 error handling. You are not obliged to construct an elaborate and hilarious tribute to some trope in popular culture, because you are an adult.
- `500.html` for critical errors, **only add generic html without template tags**
- `base.html` as entry point for `{% extends %}`

includes/

Global inclusion files should be added here, for example the navigation, django messages or tracking codes. Create additional */include* folders within addons to avoid overcrowding this directory.

Page Templates

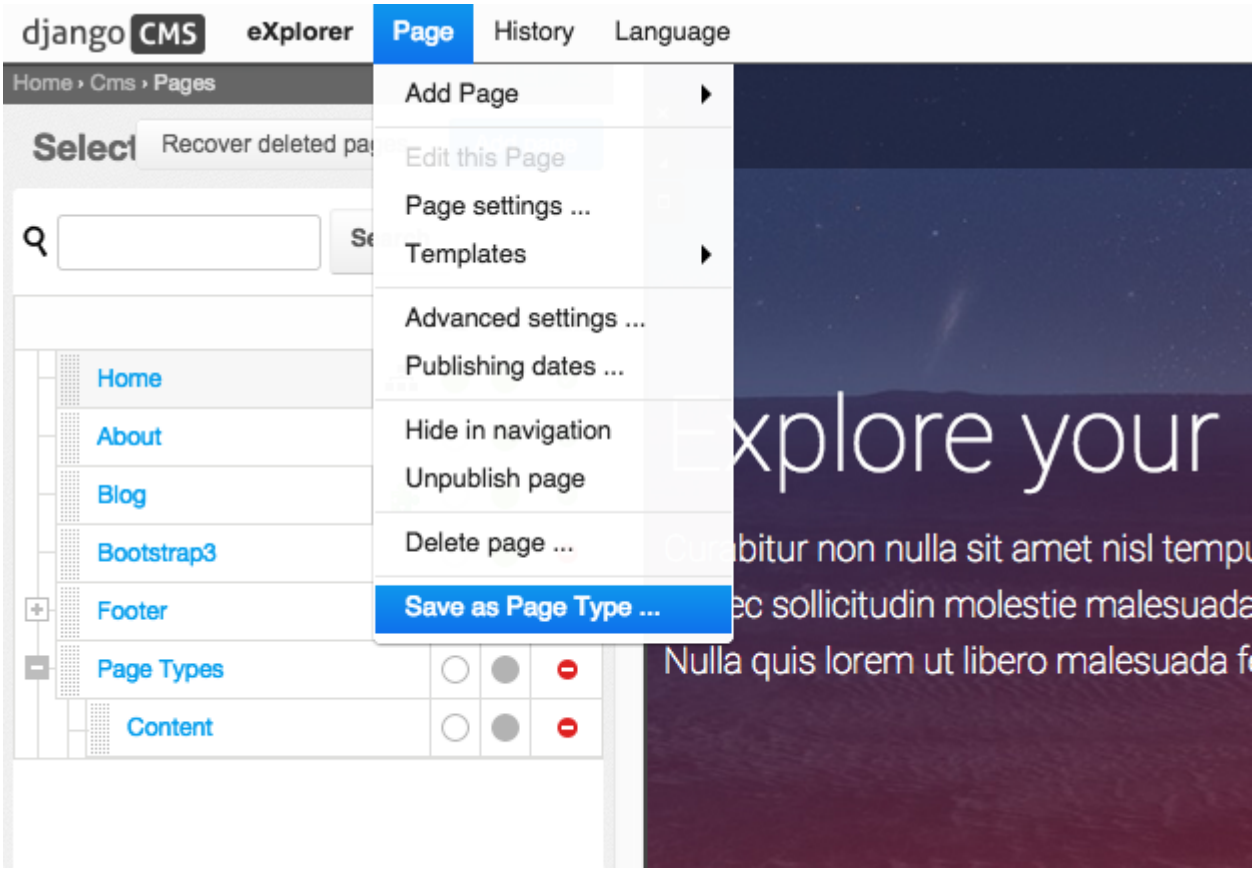
django CMS allows you to set CMS_TEMPLATES which can be chosen within the CMS by the user.



Page Types

You can save a CMS page as "Page Type" and re-use it later when creating a new page. Simply select *Page* > *Save as Page Type* .. and choose a name. You can create a new page by selecting *Page* > *Add Page* > *New Page* and choose the "Page type" you want to use. That drop down does not show up if there are no page types saved.

Page types are listed separately within the menu tree underneath *Page Types*. This allows you to change or delete them at any time if required.



Blocks and Placeholders

The content block `{% block content %}{% endblock %}` and placeholder `{% placeholder content %}` always need to be present within page templates.

Testing

Note: This section describes the unit and integration tests setup using Jasmine and Protractor. You will find advice on how to setup your own testing infrastructure, integrate it into Travis and connect with Sauce Labs.

General

We use two kinds of tests: **unit** and **integration** tests. Unit tests are simple test cases, that test a single piece of functionality within a given JavaScript file in an isolated environment without the DOM. Integration tests test the users interaction following certain move, click and keyboard interactions.

This testing infrastructure includes them both with Jasmine as the test suite and Karma as the test runner for unit tests. Protractor serves as the integration tests framework. Both tests can be run separately as described in *Commands* below.

All tests are located within `/tests`. Each pull request is validated on Travis, which runs the test executing the `gulp tests` command. You can also run this command locally, if you followed the Installation instructions.

Commands

The following commands are available to you:

- `gulp tests` runs the entire test suite
- `gulp tests:unit` only runs the unit tests
- `gulp tests:integration` only runs the integration tests
- `gulp tests:watch` to start karma to watch unit tests

Naming

The naming for tests should adhere to the conventions established in General and JavaScript.

Unit tests should be prefixed using `test` before the name file name and **integration tests** use `spec`. For example:

```
test.header.js
test.footer.js
test.content.typography.js
test.content.wysiwyg.js
...
```

```
spec.header.js
spec.footer.js
spec.content.typography.js
spec.content.wysiwyg.js
...
```

Structure

Unit tests are located within `/tests/unit` and integration tests within `/tests/integration` to create a clear separation. There are several configuration files available within the `/tests` directory described in Unit Tests and Integration Tests respectively.

The starting structure looks like this:

```
tests/
- fixtures/
- integration/
- unit/
- base.conf.js
- karma.conf.js
- protractor.conf.js
```

Fixtures and coverage are described in more depth within Unit Tests.

Configuration

The configuration files are located at the root of the `/tests` folder. `karma.conf.js` defines the settings for the `gulp tests:unit` command and `protractor.conf.js` for the `gulp tests:integration` command.

The function of these configuration files is described in more depth within Unit Tests or Integration Tests.

Browserslist

Browserslist enables us to provide a compiled and ready to use browser-list to services such as Sauce Labs, Autoprefixer and more.

Simply add the required browser to the `browserslist` file. Our configuration includes the *last 2 versions* and *ie >= 9*.

Local Server

You need to be able to run django to start a local server:

- run `cd tools/server`
- run `make install` to setup the server
- run `make run` to start the server

the development server will be reachable on `http://0.0.0.0:8000/`

Unit Tests

Configuration

The main configuration file to look at is `/tests/karma.conf.js`. It configures our **files**, **exclude** and **preprocessors** paths.

In **files** you add all the files required to be loaded in your tests. For example if we do not add jQuery into this array, we would not have it available to us while the tests run.

The **exclude** setting allows us to specifically exclude certain files from loading in the browser. By defining a path - say `addons/*.js` in `exclude` - we can exclude scripts such as `addons/myscript.js`.

Finally, add all files you want to be covered to **preprocessors**. We do not simply include all files, as we cannot guarantee the coverage of libraries or 3rd party addons.

Fixtures

`/tests/fixtures` is used to load HTML snippets in your **unit tests**. Simply define the path in test file, load the fixture and then test against it.

```
// load the fixture
fixture.setBase('tests/fixtures');
this.markup = fixture.load('snippet.html');

// test
expect(fixture.el.firstChild).to.equal(this.markup[0][0]);

// now let's cleanup
fixture.cleanup()
```

You can find more information about this in the karma-fixtures documentation.

Coverage

This folder is added when running **unit tests** either through `gulp tests`, `gulp tests:unit` or `gulp tests:watch`. Coverage uses the istanbul tool to give you a nice UI for debugging. Just simply launch the *index* file in either one of the sub-folders generated. There can be as many sub-folders as clients connected to your runner.

It's worth to mention that the success of your project does not depend on the tests or the percentage of your code coverage, but it will improve maintenance and further development for you and other contributors. We should aim for the highest possible coverage.

Integration Tests

Configuration

The main configuration file to look at is `/tests/protractor.conf.js`. It configures our `browserName`.

In `browserName` we specify the browser that will be used to launch the tests. It can be set to `phantomjs`, `firefox` or `chrome`.

You can find more information about this in the protractor referenceConf.js documentation.

All spec files should be placed in `/tests/integration/specs` and all page object files should be in `/tests/integration/pages`. So, the file organisation structure is:

```
tests/
- integration/
   - specs/
   |   - spec.name.js
   |   - spec.another.name.js
   - pages/
       - page.name.js
       - page.another.name.js
```

The specs that will be launched are defined in the `gulpfile.js`. They can be specified using patterns:

```
return gulp.src([PROJECT_PATH.tests + '/integration/specs/*.js'])
```

By default all specs inside `/tests/integration/specs` folder will be launched.

Coverage

Integration coverage is measured by the number of critical path or regression test cases that were automated. Keep in mind that the success of your project does not depend on the tests or the percentage of your code coverage, but it will improve maintenance and give you and other contributors more confidence in the quality of the product you produce. We should aim for the highest possible coverage and quality.

Services

Travis

The Travis configuration is fairly straightforward. You can see our example configuration file for reference. The important point here is to add all your credentials using `travis encrypt` for security reasons.

You can install the `travis` command line tool by running `gem install travis`.

Sauce Labs

Sauce Labs helps us to run our unit and integration tests on multiple browsers.

When using our test suite locally, phantomjs is used in the interests of speed, especially on integration tests. However this does not test on real browsers, so is not comprehensive and various issues can slip through undetected. In order to provide real-browser test coverage, `.travis.yml` is configured to connect to Sauce Labs and run our test against a matrix of browsers.

For each new setup you need to adapt the `env: global:` variables by adding:

```
travis encrypt SAUCE_USERNAME={USER} --add
travis encrypt SAUCE_ACCESS_KEY={TOKEN} --add
```

Where `{USER}` represents the sub-account user name and `{TOKEN}` the sub-account token.

See the example .travis.yml:

- the first `secure` line in `env: global:` represents encrypted Sauce Labs sub-account user name
- the second `secure` line stands for encrypted Sauce Labs sub-account token
- the third `secure` line is the encrypted Code Climate token

Important: To get the correct status image from Sauce Labs you will have to create **sub-accounts** for each project. Otherwise, all tests will share the same badge.

We set up the configuration files to skip Sauce Connect when you test locally; these tests will only run on Travis.

Browser Matrix

You can configure the browser matrix within `/tests/base.conf.js`. There is an elegant platform configurator available to you if you want to add more browsers.

Code Climate

We also support Code Climate to show the current coverage status. You simple need to import your project and add the appropriate repo token:

```
travis encrypt CODECLIMATE_REPO_TOKEN={TOKEN} --add
```

Where `{TOKEN}` represents the key from Code Climate.

Coveralls

You can use Coveralls as an alternative to show the current coverage status. You simply need to import your project and Karma will take care of the rest.

Coding Style

Note: Compare to the Guidelines and the Structure we offer many more coding conventions, most of them are being covered by certain linting tools such as ESLint and CSS Lint. You will also find good practice and common sense here. Yet note that these are conventions and eventually will make their way to the guidelines.

JavaScript

Note: This section is heavily inspired by a Airbnb JavaScript Style Guide, Yandex Codestyle, Idiomatic Javascript and lots of common sense, really.

Why?

> "All code in any code-base should look like a single person typed it, no matter how many people contributed." - Rick Waldron

These are the fundamental principles we should follow when we design and develop software.

- Consistent code is easy to read.
- Simple code is easy to maintain.
- In simple expressions it's harder to make mistakes.

Formatting

Blocks

Use braces with all blocks. Don't do inline blocks.

```
// bad
if (test)
    return false;

// bad
if (test) return false;

// good
if (test) {
    return false;
}

// bad
function () { return false; }

// good
function () {
    return false;
}
```

When you're using multi-line blocks with if and else, put else on the same line as your if block's closing brace.

```
// bad
if (test) {
  thing1();
  thing2();
}
else {
  thing3();
}
```

```
// good
if (test) {
  thing1();
  thing2();
} else {
  thing3();
}
```

Comments

Follow the guidelines. Use `//` for single line comments. Place single line comments on a newline above the subject of the comment. Between the `//` and the text of the comment should be one space character.

```
// bad
var active = true;  //is current tab

// good
// is current tab
var active = true;
```

Most importantly, **keep comments up to date** if the code changes.

Whitespace

With proper `.editoconfig` and `eslint` setup these will be enforced automatically, but still:

- 4 spaces for tabs.
- Place 1 space before leading curly brace.
- Place 1 space before the opening parenthesis in `if`, `while`, etc.
- Place 1 space after colon.
- Place no space before the argument list in function calls and declarations, e.g. `function fight() { ... }`
- Set off operators with spaces, e.g. `var x = 2 + 2;`
- No whitespace at the end of line or on blank lines.
- **Lines should be no longer than 120 characters. There are 2 exceptions, both allowing the line to exceed 120 characters:**
  - If the line contains a comment with a long URL.
  - If the line contains a regex literal. This prevents having to use the regex constructor which requires otherwise unnecessary string escaping.
- End files with a single newline character.

Use indentation when making long method chains. Use a leading dot, which emphasises that the line is a method call, not a new statement.

```
// bad
$('#items').find('.selected').highlight().end().find('.open').updateCount();

// bad
$('#items').
```

```
    find('.selected').
        highlight().
        end().
    find('.open').
        updateCount();

// good
$('#items')
    .find('.selected')
        .highlight()
        .end()
    .find('.open')
        .updateCount();
```

Leave a blank line after blocks and before the next statement

```
// bad
if (foo) {
    return bar;
}
return baz;

// good
if (foo) {
    return bar;
}

return baz;

// bad
var obj = {
    foo: function() {
    },
    bar: function() {
    }
};
return obj;

// good
var obj = {
    foo: function() {
    },

    bar: function() {
    }
};

return obj;
```

Use newlines to group logically related pieces of code. For example:

```
doSomethingTo(x);
doSomethingElseTo(x);
andThen(x);

nowDoSomethingWith(y);

andNowWith(z);
```

Commas

- Leading commas: God, **no**!
- Additional trailing comma: **No**

```
// bad
var hero = {
    firstName: 'Kevin',
    lastName: 'Flynn',
};

var heroes = [
    'Batman',
    'Superman',
];

// good
var hero = {
    firstName: 'Kevin',
    lastName: 'Flynn'
};

var heroes = [
    'Batman',
    'Superman'
];
```

Semicolons

Yes, always.

```
// bad
(function () {
    var name = 'Skywalker'
    return name
})()

// good
(function () {
    var name = 'Skywalker';
    return name;
})();

// good (guards against the function becoming an argument when two files
// with IIFEs are concatenated) this should not happen if the previous
// example is enforced, but sometimes we have no control over vendor code
;(function () {
    var name = 'Skywalker';
    return name;
})();
```

Variables

General

Always use var to declare variables. Not doing so will result in global variables. We want to avoid polluting the global namespace

Assign variables at the top of their scope. This helps avoid issues with variable declaration and assignment hoisting related issues.

Use one var declaration per variable. It's easier to add new variable declarations this way, and you never have to worry about swapping out a `;` for a `,` or introducing punctuation-only diffs.

```
// bad
var items = getItems(),
    goSportsTeam = true,
    dragonball = 'z';

// bad
// (compare to above, and try to spot the mistake)
var items = getItems(),
    goSportsTeam = true;
    dragonball = 'z';

// good
var items = getItems();
var goSportsTeam = true;
var dragonball = 'z';
```

Objects

Use the literal syntax for object creation.

```
// bad
var item = new Object();

// good
var item = {};
```

Don't use reserved words as keys.

```
// bad
var superman = {
    default: { clark: 'kent' },
    private: true
};

// good
var superman = {
    defaults: { clark: 'kent' },
    hidden: true
};
```

Do not use quotes for properties, it is only needed for screening reserved words which we are not supposed to use.

Arrays

Use the literal syntax for array creation.

```
// bad
var items = new Array();

// good
var items = [];
```

Use `Array#push` instead of direct assignment to add items to an array.

```
var someStack = [];

// bad
someStack[someStack.length] = 'abracadabra';

// good
someStack.push('abracadabra');
```

To convert an array-like object to an array, use `Array#slice`. If you need to copy an array, use slice as well.

```
function trigger() {
    var args = Array.prototype.slice.call(arguments);
    ...
}
```

```
var length = items.length;
var itemsCopy = [];
var index;

// bad
for (index = 0; index < length; index++) {
    itemsCopy[index] = items[index];
}

// good
itemsCopy = items.slice();
```

Strings

Use single-quotes for strings. When programmatically building a string use `Array#join` instead of string concatenation

```
// bad
var template = '<div class="whatever">' +
    message +
'</div>';

// good
var template = [
    '<div class="whatever">',
        message,
    '</div>'
].join('');
```

If you have a complicated string buildup it's always better to use javascript templating instead. That way templates could have their own files with proper syntax highlighting and pre-compilation build step.

Functions

Function expressions:

```
// anonymous function expression
var anonymous = function () {
    return true;
};

// named function expression
var named = function named() {
    return true;
};

// immediately-invoked function expression (IIFE)
(function () {
    console.log('Welcome to the Internet. Please follow me.');
})();
```

Tend to avoid anonymous function expressions, try to always use named ones, it will save you a lot of pain going through stack traces and debugging in general.

Never declare a function in a non-function block (if, while, etc). Assign the function to a variable instead. Browsers will allow you to do it, but they all interpret it differently, which is really bad news.

```
// bad
if (currentUser) {
    function test() {
        console.log('Nope.');
    }
}

// good
var test;
if (currentUser) {
    test = function test() {
        console.log('Yup.');
    };
}
```

Never name a parameter arguments. This will take precedence over the arguments object that is given to every function scope. It is also a reserved word.

```
// bad
function nope(name, options, arguments) {
    // ...stuff...
}

// good
function yup(name, options, args) {
    // ...stuff...
}
```

Prefer early returns.

```
// bad
function returnLate(foo) {
    var value;
```

```
    if (foo) {
        value = 'foo';
    } else {
        value = 'quux';
    }
    return value;
}

// good

function returnEarly(foo) {
    if (foo) {
        return 'foo';
    }

    return 'quux';
}
```

```
// bad
function doThingsWithComponent(element) {
    if (element.length) {
        // do things
    }
}

// good
function doThingsWithComponent(element) {
    if (!element.length) {
        return false;
    }

    // do things
}
```

Functions context

Prefer `Function#bind` over `$.proxy(function (), scope)`.

```
doAsync(function () {
    this.fn();
}.bind(this));
```

If the context argument is available, it is preferred.

```
// bad
[1, 2, 3].forEach(function (number) {
    this.fn(number);
}.bind(this));

// good
[1, 2, 3].forEach(function (number) {
    this.fn(number);
}, this);
```

If assigning the current context to a variable, the variable should be named `that`:

```
var that = this;
doAsync(function () {
    that.fn();
});
```

Properties

Use dot notation when accessing properties.

```
var luke = {
    jedi: true,
    age: 28
};

// bad
var isJedi = luke['jedi'];

// good
var isJedi = luke.jedi;
```

Use subscript notation `[]` **only** when accessing properties with a variable.

```
var luke = {
    jedi: true,
    age: 28
};

function getProp(prop) {
    return luke[prop];
}

var isJedi = getProp('jedi');
```

Hoisting

Variable declarations get hoisted to the top of their scope, but their assignment does not.

```
// we know this wouldn't work (assuming there
// is no notDefined global variable)
function example() {
    console.log(notDefined); // => throws a ReferenceError
}

// creating a variable declaration after you
// reference the variable will work due to
// variable hoisting. Note: the assignment
// value of `true` is not hoisted.
function example() {
    console.log(declaredButNotAssigned); // => undefined
    var declaredButNotAssigned = true;
}

// The interpreter is hoisting the variable
// declaration to the top of the scope,
// which means our example could be rewritten as:
function example() {
```

```
    var declaredButNotAssigned;
    console.log(declaredButNotAssigned); // => undefined
    declaredButNotAssigned = true;
}
```

Anonymous function expressions hoist their variable name, but not the function assignment.

```
function example() {
    console.log(anonymous); // => undefined

    anonymous(); // => TypeError anonymous is not a function

    var anonymous = function() {
        console.log('anonymous function expression');
    };
}
```

Named function expressions hoist the variable name, not the function name or the function body.

```
function example() {
    console.log(named); // => undefined

    named(); // => TypeError named is not a function

    superPower(); // => ReferenceError superPower is not defined

    var named = function superPower() {
        console.log('Flying');
    };
}

// the same is true when the function name
// is the same as the variable name.
function example() {
    console.log(named); // => undefined

    named(); // => TypeError named is not a function

    var named = function named() {
        console.log('named');
    }
}
```

Function declarations hoist their name and the function body.

```
function example() {
    superPower(); // => Flying

    function superPower() {
        console.log('Flying');
    }
}
```

For more information on hoisting refer to JavaScript Scoping & Hoisting by Ben Cherry.

Types

Type Casting and Coercion

Strings:

```
//  => this.reviewScore = 9;

// bad
var totalScore = this.reviewScore + '';

// good
var totalScore = '' + this.reviewScore;

// bad
var totalScore = '' + this.reviewScore + ' total score';

// good
var totalScore = this.reviewScore + ' total score';
```

Numbers: Use parseInt for `Numbers` and always with a radix for type casting.

```
var inputValue = '4';

// very bad
var val = new Number(inputValue);

// bad
var val = +inputValue;

// bad
var val = inputValue >> 0;

// bad
var val = parseInt(inputValue);

// ok
var val = Number(inputValue);

// good
var val = parseInt(inputValue, 10);
```

Booleans:

```
var age = 0;

// bad
var hasAge = new Boolean(age);

// ok
var hasAge = Boolean(age);

// good
var hasAge = !!age;
```

Comparison Operators & Equality

Use === and !== over == and !=.

Comparison operators are evaluated using coercion with the ToBoolean method and always follow these simple rules:

- **Objects** evaluate to **true**
- **Undefined** evaluates to **false**
- **Null** evaluates to **false**
- **Booleans** evaluate to the **value of the boolean**
- **Numbers** evaluate to **false** if **+0**, **-0**, or **NaN**, otherwise **true**
- **Strings** evaluate to **false** if an empty string '', otherwise **true**

```
if ([0]) {
    // true
    // An array is an object, objects evaluate to true
}
```

- Use shortcuts.

```
   // bad
   if (name !== '') {
     // ...stuff...
   }

   // good
   if (name) {
     // ...stuff...
   }

   // bad
   if (collection.length > 0) {
     // ...stuff...
   }

   // good
   if (collection.length) {
     // ...stuff...
   }

More info in `Javascript Equality Table <https://dorey.github.io/JavaScript-Equality-Table/>`_
```

- Condition statements should not contain assignment operations:

```
// bad
var foo;
if ((foo = bar()) > 0) {
    // ...
}

// good
var foo = bar();
if (foo > 0) {
    // ...
}
```

- Logical operators should not be used for conditional branching:

```
// bad
condition && actionIfTrue() || actionIfFalse();

// good
if (condition) {
```

```
    actionIfTrue();
} else {
    actionIfFalse();
}
```

- Conditions longer than the maximum line length should be divided as in the example:

```
// good
if (longCondition ||
    anotherLongCondition &&
    yetAnotherLongCondition
) {
    // ...
}
```

- The ternary operator should be written as in the examples:

```
var x = a ? b : c;

var y = a ?
    longButSimpleOperandB : longButSimpleOperandC;

var z = a ?
    moreComplicatedB :
    moreComplicatedC;
```

- If a statement is longer than the maximum line length, it is split into several lines and properly indented.
- Closing parentheses should be on a new line with the indentation of the current block statement. Tend to do the same with object properties.

```
DoSomethingThatRequiresALongFunctionName(
    veryLongArgument1,
    argument2,
    argument3,
    argument4
);
anotherStatement;
```

jQuery

Variables

Do not prefix jQuery variables with `$`. Always cache jQuery lookups.

```
// bad
function setSidebar() {
    $('.sidebar').hide();
    $('.sidebar').css({
        'background-color': 'pink'
    });
}

// bad
function setSidebar() {
    var $sidebar = $('.sidebar');
    $sidebar.hide();
    $sidebar.css({
```

```
        'background-color': 'pink'
    });
}

// good
function setSidebar() {
    var sidebar = $('.sidebar');
    sidebar.hide();
    sidebar.css({
        'background-color': 'pink'
    });
}
```

Ajax

Prefer promise based `$.ajax` calls over callback passing into settings object.

```
// bad
$.ajax('/url', {
    dataType: 'json',
    success: function () {
    },
    error: function () {
    },
    complete: function () {
    }
});

// good
$.ajax({
    urls: '/url',
    dataType: 'json',
}).done(function myAjaxDone () {
    ...
}).fail(function myAjaxFailed () {
    ...
}).always(function myAjaxIsCompleted () {
    ...
});
```

The nice thing about this is that the return value of `$.ajax` is now a deferred promise that can be bound to anywhere else in your application. So let's say you want to make this ajax call from a few different places. Rather than passing in your success function as an option to the function that makes this ajax call, you can just have the function return $.ajax itself and bind your callbacks with done, fail, then, or whatever. Note that `always` is a callback that will run whether the request succeeds or fails. `done` will only be triggered on success.

It is also easier to process when you need to pass multiple success callbacks with few chained *.done* calls (which can also be conditional) than passing array of functions into `success` property.

```
...
getItems: function getItems(options) {
    var opts = $.extend({
        url: '/items/',
        dataType: 'json',
        ...
    }, options);
    return $.ajax(opts);
}
```

```
...

// and then in the app
this.getItems().done(function (products) {
    ...
})

// and in all the different places
this.getItems({ url: '/items/categories/12' }).done(function (products) {
    ...
});
```

Common patterns

Loops

Use `for-in` only for iterating over keys in an `Object`, never over an `Array`.

Naming conventions

Refer to guidelines. Use leading underscore to denote private methods/properties. The only place where it's allowed to use single letter variable is in event callbacks:

```
// bad
$('div.elem').on('click', function (clickEvent) {
    ...
});

// good
$('.js-element').on('click', function (e) {
    ...
});
```

Events

When attaching data payloads to events (whether DOM events or something more proprietary like Backbone events), pass a hash instead of a raw value. This allows a subsequent contributor to add more data to the event payload without finding and updating every handler for the event. For example, instead of:

```
// bad
$(this).trigger('listingUpdated', listing.id);

...

$(this).on('listingUpdated', function(e, listingId) {
    // do something with listingId
});
```

prefer:

```
// good
$(this).trigger('listingUpdated', { listingId: listing.id });

...
```

```
$(this).on('listingUpdated', function(e, data) {
    // do something with data.listingId
});
```

Templates

When passing data to JS templates (using underscore.js / window.tmpl by J. Resig) - always pass an object that has only one property, and that property is the data you need.

Consider this template:

```
<% if (people) { %>
   <%= people %>
<% } %>
```

```
// bad
var markup = tmpl(template, { prop1: true, prop2: '1' });
```

This will throw a ReferenceError because these template engines use `with` underneath. Instead do this:

```
<% if (addon.people) { %>
   <%= addon.people %>
<% } %>
```

```
// good
var markup = tmpl(template, {
    addon: {
        prop1: true,
        prop2: '1'
    }
});
```

You will have explicit scope without any unexpected behaviours.

Classes

It is a common pattern when creating javascript components to save all the ui elements under a common namespace. It is also a common mistake to declare an object called `ui` on a class.

```
// bad
var Widget = new Class({
    ui: {
        oneElement: null,
        anotherElement: null
    },
    initialize: function (container, options) {
        this._buildUI(container);
    },
    _buildUI: function (container) {
        this.container = $(container);

        // another bad thing
        this.ui.oneElement = $('.js-one-element');
        this.ui.anotherElement = $('.js-another-element');
    }
});
```

There are several problems. The `ui` object is declared on prototype in this case, and as with all complex types is javascript we are working with a reference to the value. That means that the same ui object will be shared across all instances of the class, which in turn will mean that you won't be able to use several instances on the page.

```
// good
var Widget = new Class({
    initialize: function (container, options) {
        this._buildUI(container);
    },
    _buildUI: function (container) {
        this.container = $(container);
        this.ui = {
            // scoping widget's moving parts under the same container is a good pattern as well
            oneElement: $('.js-one-element', this.container),
            anotherElement: $('.js-another-element', this.container)
        };
    }
});
```

We do not always know how the widget will be used. Even if "it's only gonna be on this page and it's gonna be this particular instance" seems like a valid reason not to change - it never is. We should always strive for making components independent and reusable, it's usually not a big effort (especially if you think about before writing the widget) and it can solve a lot of problems for you in the future.

Passing data to components

Avoid instantiating components in inline scripts. Instead pass the data to the components through data attributes.

Avoid spreading options into multiple data attributes, as it might happen that two different javascript components live on the same DOM node and require an option with the same name. Instead use json notation.

Bad:

```
<div class="js-component-1 js-component-2"
    data-something="false" {# for component 2 #}
    data-value="for component 1"
    data-value="for component 2"> {# aw maaan #}
    Sad panda :(
</div>
```

Imagine in this case component 1 functionality is significantly affected by an option that is meant for component 2. Also if they share the same option property name, such as value - sad panda.

Good:

```
<div class="js-component-first js-component-second"
    data-component-first='{
        "value": "for component 1"
    }'
    data-component-second='{
        "value": "for component 2",
        "something": false
    }'>
    Happy panda!
</div>
```

Passing the data to the components is also very straightforward. This way you have the same initialisation method for all existing instances of the widget even if they have different options.

```
var componentElements = $('.js-component-2');
var defaults = {
    x: 0,
    y: 0,
    something: true
};
componentElements.each(function () {
    var componentElement = $(this);
    var options = $.extend({}, defaults, componentElement.data('component-second'));
    new ComponentSecond(componentElement, options);
}):
```

Magic numbers

- Avoid magic numbers. Try to parametrise or use constants.

```
// bad
setTimeout(function () {
    if (failed && count < 5) {
        count++;
        return;
    }
    // or do stuff
}, 3000);

// better
var POLLING_TIMEOUT = 3000;
var MAX_FAILURES_COUNT = 5;

setTimeout(function () {
    if (failed && count < MAX_FAILURES_COUNT) {
        count++;
        return;
    }
    // or do stuff
}, POLLING_TIMEOUT);
```

```
switch (e.keyCode) {
    case keyCodes.ENTER:
    case keyCodes.SPACE:
        x();
        break;
    case keyCodes.TAB
    case keyCodes.ESCAPE:
        y();
        break;
    default:
        z();
}
```

ECMAScript 5

Use where appropriate. Use array methods for working with arrays, but don't use them when working with array-like objects such as jQuery collections. For them use `$.fn.each` instead.

Prefer `Array#forEach` over `for () {}` loop.

```
var fighters = [
    {
        name: 'Jonny Cage',
        dead: true
    },
    {
        name: 'Kung Lao',
        dead: true
    },
    {
        name: 'Raiden',
        dead: false
    }
];

// bad
var i;
var l = fighters.length;

for (; i < l; i++) {
    console.log(fighters[i].name + ' ' + (fighters[i].dead ? 'lost' : 'did not lose'));
}

// good
fighters.forEach(function (fighter) {
    console.log(fighter.name + ' ' + (fighter.dead ? 'lost' : 'did not lose'));
});
```

More info on ES5 compatibility here

Styles

General

Formatting, nesting, ordering and everything else is covered by Guidelines

Main problem with CSS

> There are two types of problems in CSS: cosmetic problems and architectural problems. Cosmetic problems—issues like vertical centering or equal-height columns—usually engender the most vocal complaints, but they're almost never showstoppers. They're annoying, sure, but they don't break the build.
>
> Philip Walton Side Effects in CSS

Since CSS is global, every rule you write or override has the potential to break completely unrelated things. With that in mind, try to avoid selectors that are too unspecific (e.g. type selectors or overly specific selectors, like `.nav > ul > li > a`. That selector is going to be extremely painful to extend and override if there's going to be a "special" list item for example. That also brings us to

Selector performance

It is always said that css selectors performance is not that important and there are no "easy-to-follow" rules for fixing it. But just to reiterate, main points:

- If your project is sufficiently big and complex or really dynamic, css selector performance may play a major role in the perceived rendering performance.
- Selectors are interpreted by the browser from right to left, meaning `.my-class > *` will select all the elements on the page all the time and check if their immediate parent has a class `my-class`. If there would be no `>` it would traverse the tree all the way up for every element, which is not very good. It is true that browsers do optimize things like this, but you should always check for yourself.

JS selectors

We use `js-` prefixed selectors for referencing DOM Nodes from javascript. That means that these classes have a pure functional purpose and styles should **never** be applied to them. Same type of widget could be easily represented by completely different sets of markup.

Magic numbers

Tend not to use magic numbers in CSS. Let's say you want to position an element in a specific place. Try to be agnostic of the environment and don't use values that are too specific.

```
.nav {
    height: 30px;
}

// bad
.dropdown {
    // it works, but imagine we are going to change
    // the height of the nav. we'll need to go all over the css and change
    // the value now
    top: 35px;
}

// good
.dropdown {
    top: 100%;
    margin-top: 5px;
}
```

Another example of magic numbers could be computed values. Let's say you have a component that is created on top of existing component, like a bootstrap styled select.

```
// bad
.custom-select {
    height: 38px;
    padding: 14px 17px;
}

// much better
.custom-select {
    height: $input-height-base - 2px;
    padding: ($padding-base-vertical - 1px) ($padding-base-horizontal - 1px);
}
```

Avoid magic numbers like the plague..

Sass

Sass or SCSS

That one is a no-brainer. We use SCSS flavor because it is closer to CSS and easier to pick up for everyone. It also resolves subtle issues with indentation.

Nesting

Optimal nesting level is 2. You can go up to 4 levels (scss-lint rule), but try not to. Overused nesting usually means that something is wrong with the code.

Extends

In general, try to avoid extend unless you know exactly what you are doing. Only use @extend when the rulesets that you are trying to DRY out are inherently and thematically related.

> Do not force relationships that do not exist: to do so will create unusual groupings in your project, as well as negatively impacting the source order of your code.
>
> http://csswizardry.com/2014/11/when-to-use-extend-when-to-use-a-mixin/

Color manipulation

When using alpha transparent colors keep in mind that `rgba` supports passing colors, so you can do things like this:

```
// bad
color: rgba(0, 0, 0, 0.85);

// good
color: rgba(black, 0.85);
color: rgba(#000, 0.85);
color: rgba($color, 0.85);
```

Autoprefixer

For generating vendor prefixes one should use Autoprefixer instead of relying on mixins. That way we reduce sass compilation time and ensure that we have only prefixes that we actually need. As a good side effect we will use actual standard CSS syntax.

Bootstrap

When using `settings/_bootstrap.scss` make sure that you have all the variables overwritten in the file, because overriding only some of them can lead to subtle bugs like this:

```
// this is what happens in the bootstrap/_variables.scss
$line-height-computed: 20px !default;
$padding-base-vertical: 6px !default;

// and this is a computed property from bootstrap, 34px by default
$input-height-base:   ($line-height-computed + ($padding-base-vertical * 2) + 2) !default;

// now what we want to do is to override line-height-computed in our settings file
$line-height-computed: 23px;
```

Now we would expect that `$input-height-base` will be 37px, but it will be still 34px because computed properties are already calculated and won't be changed. Since bootstrap components dimensions are all interconnected to these computed variables we should always have the full settings file. Order matters too.

Media queries

In general when using media queries with bootstrap variables, use appropriate values for appropriate type of a query.

```
// bad
@media (min-width: $screen-sm-max) {
    ...
}

@media (max-width: $screen-sm-min) {
    ...
}

// good
@media (min-width: $screen-md-min) {
    ...
}

@media (max-width: $screen-xs-max) {
    ...
}
```

These values differ only by 1 pixel, but it's a very important one.

Open for discussion

- Screenshot regression testing
- autoprefixer implementation

Tips and Tricks

Note: There are several tips & tricks we found over the time that are worth mentioning.

Floating

When using `float: left;`, `display: block;` is not required anymore as **every** element which is floated automatically gets the **block** state. This does not apply to sub-elements.

Hidden Attribute

With modern HTML5 we can use the `hidden="hidden"` attribute which is a **softer** version of `display: none;`. This state can easily be overwritten using CSS or JavaScript. As such the attribute is ideal for hiding elements which are later displayed through JavaScript to prevent jumping behaviours. But be aware of the current support.

Image Optimisation

Images are the number one source of optimisation when it comes to file size. Optimise images using tools like CodeKit, ImageOptim or our internal Gulp command: `gulp images`.

Contribution

Note: You are very welcome improving this boilerplate for Aldryn and your everyday use, especially the documentation always needs love. Feel free to fork and send us pull requests and follow the guidelines from within this section.

Code of Conduct

- Ensure code validates against or own guidelines
- Write documentation about what you are doing
- If you are not sure, just ask - join our community **#aldryn** on Freenode

Documentation

To extend and run the documentation, you will need Python and Virtualenv installed on your computer. You also need Git and a GitHub account obviously.

In addition, follow the steps underneath to get them running:

1. clone the repository using `git clone https://github.com/aldryn/aldryn-boilerplate-bootstrap3.git`
2. navigate to the documentation through `cd aldryn-boilerplate-bootstrap3/docs`
3. run `make install` to install additional requirements
4. run `make run` to let the server run

Now you can open **http://localhost:8000** on your favourite browser and start changing the rst files within `docs/`.

You need to be aware of reStructuredText to format the documentation properly.

Guidelines

- Always start paths with a `/` and leave the trailing slash.
- Leave two spaces before a title.
- Write "Django", "django CMS" or "Aldryn".
- Write names properly: Sass, Bootstrap, JavaScript instead of sass (or SASS), bootstrap and javascript.
- Additional guidelines from django CMS apply.

Pull Requests

Before starting to work on issues or features, please mind the branching model:

- **master** is used for hotfix releases (1.1.x)

- **develop** is used for features and issues (1.x.x)

Everything that is merged to *develop* will be released within the next proper release (1.x.x). Major releases (x.0.0) will have their own branches but are always merged to *develop* before releasing to master.

A pull request needs the consent of two developers familiar with this repository to be merged.

Releases

- Adapt the CHANGELOG.rst
- Adapt AUTHORS.rst if required
- Bump version in boilerplate.json
- Create a GitHub tag
- Add the release notes on the GitHub tag
- Build new tag on readthedocs.org
- Run `bash tools/release.sh` before release on Aldryn
- Run `aldryn boilerplate upload` to release on Aldryn
- Test, inform, present