Aldryn Boilerplate Standard Documentation

Release 3.0.5

Divio AG

September 14, 2015

Contents

The most advanced **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. In addition we implement various best practices from within the front-end community.

This boilerplate is compatible with Aldryn.

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

CHAPTER 1

Documentation

1.1 General

This chapter provides general information about:

1.1.1 Requirements

The following software should be installed on your system in order to use this boilerplate:

- Sass: http://sass-lang.com/
- Compass: http://compass-style.org/
- Bootstrap: https://github.com/twbs/bootstrap-sass

You can compile/watch using `compass watch private` from within the root.

Optional

We provide some automation using the Gulp task runner. You will need the following requirements in order to use it:

- Node JS: http://nodejs.org/
- Node Package Manager: https://www.npmjs.org/
- Bower: http://bower.io/
- Gulp JS: http://gulpjs.com/

After all requirements are met, install the packages using the `npm install` command from within the boilerplate's root. You can install the bower requirements by running `bower install`.

You can run **Gulp** commands from within your base folder using `gulp`. If you would like to run specific tasks, consult the **gulpfile.js** within the base folder.

1.1.2 Best Practices

Naming

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

As you can spend hours in scheming name patterns, structure and conventions we only advice to follow the BEM principles but using as a separator **one dash** only and always lowercase format: `blockname-elementname`.

Automation

We try to make our live as easy as possible. For this reason we implemented Gulp JS as task runner instead of Grunt as we prefer **code over configuration**. There are some helpful commands available:

- `gulp` runs gulp lint browser and watch commands
- `gulp lint` lints all JavaScript using `.jshintrc` and `.jscsrc`
- `gulp images` optimised images within `/static/img`
- `gulp browser` connects to a given server (django) and runs livereload on `http://0.0.0.0:3000`
- `gulp watch` starts a watch command for linting

Bootstrap Plugins

We are implementing the following additional Bootstrap plugins into the setup:

- Select2
- Cl.Debug

Browsers

In order to display an automated message when JavaScript is disabled or there might be lack of support, we integrated the Outdated Browser script within this boilerplate. Styles and settings are automatically set from within Bootstrap.

Editors

You can use any editor you want, to make your life a bit easier we implemented EditorConfig into the boilerplate's root `.editorconfig`.

Icons

We integrated the Font Awesome library instead of Bootstrap's Glyphicons and stream the `fa-*` prefixes to `icon-*` to be more consistent and flexible when using fontastic.me.

Libraries

We are implementing the following standard libraries in addition to the default requirements from Bootstrap:

- Class JS
- Respond JS
- SWF Object
- HTML5 Shiv

We implemented Bower to help you manage dependencies. Packages are automatically downloaded into `/static/vendor/` but **not** moved to their appropriate folders. This still requires manual work.

Tests

We currently implemented a basic test framework within `static/js/tests` using QUnit. YOu can simply run tests using the Gulp command `gulp tests`.

1.1.3 Folder Structure

The generic folder structure is as follows:

private/

This folder is intended for placing preprocessing libraries such as **sass**, **coffeescript** or **haml**. Simply create a folder with the associated name of the library such as `sass/` and place configuration files on the same level. An example structure would look like:

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

Hint: The `config.rb` is taken from Compass which can also be used for a native sass setup. However aldryn-boilerplate-standard uses the Compass **SCSS** - format as default.

static/

All layout specific files will be placed in this folder. The main folder structure includes:

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

If folders are not required, just simply remove them. For demo content (which might be later integrated as media files) create a folder called **dummy/**, for example: `static/img/dummy/` and place those images there. The dummy folder is intended to be **removed** before a website goes live.

When a structure might get more complicated, make use of grouping and create additional folders like `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 the 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 global folder.

1.1.4 Comments

Use comments wisely. Ideally every major feature should commented in detail, but time-restraints and budget prevent this often. However it does not make sense to comment the obvious. Use specific separators to structure code that it is more readable.

Sections

The long block comment is used to separate sections, for example the mobile/tablet/desktop separation within scss or base.js.

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

Use the half version of this to separate larger modules or code blocks so its more readable and parts can be found quickly:

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

The large comment block should be exactly 120 characters long.

Otherwise use normal inline // comments or block /* comments */ whenever it is more logical.

Notes

We also support three types of comments within the code itself:

TODO:
indicates that something is still missing and needs to be done

```
// TODO: We still need to add keyboard navigation
```

INFO:
provides additional help if something might be unclear or requires additional description

```
// INFO: We had to loop twice through the element as the provided data is nested multiple times
```

DOCS:
provides a simple docs link

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

HTML

Use the django or jinja template comments rather than the native html ones in order to hide developers notes from the live production website when the HTML gets shipped.

1.2 Markup

This chapter describes in more detail what the **html markup** guidelines are, how they are structured and what the requirements are:

1.2.1 Guidelines

- Use **4 space indentation** and **not** tabs
- Use **underscores** for html file naming
- Use **double-quotes** " for all attributes including django-template tags
- Use lowercase for **all** attributes
- HTML **has to validate** using W3C guidelines
- HTML should validate the WCAG 2.0 A guidelines
- HTML should be modular and reusable, do not use easy names like "job" or "item" on top level. Use "addon-jobs" instead
- Always use space indendation after django tags such as `{% if %}`, `{% forloop %}`, `{% block ... %}` and others
- Ignore to rule on top for `{% if %}` or `{% forloop %}`
- All templates should be placed within the roots `templates/` folder
- In general **code readability first**

Example

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

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

IDs vs Classes

You should **always** use classes instead if id's. Classes represent a more OOP approach of 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 elements or for in-page navigation such as: `<label for="field-username">..</label><input type="text" id="field-username" />` or `/some/url#whats-new`

Elements

Try to keep the html structure simple and avoid unnecessary elements. It is sometimes easier to use a single div with a single class rather than multiple divs with multiple classes:

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

We don't need to add specific classes to the **h2** as we can control the inner style using `.addon-blog`. However more complicated structures such as lead, content, author, meta infos, tags can require their own class names:

```html
<div class="addon-blog">
    <h2>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><a href="#">News</a>
        <li><a href="#">Blog</a>
        <li><a href="#">Tags</a>
    </ul>
</div>
```

1.2.2 Structure

Django automatically looks for a `base.html` yet our base extends `base_root.html`. This is a good example of how Django's template inheritance is working. In order to keep the basic html structure minimalistic, we outsource all head and foot relevant code into base_html which makes this file better maintainable.

Where to start

Build your general structure within **base.html**. This includes namely the header and the footer. Do not split up header and footer into separate files, you can use django blocks and overwrite default behaviours when needed. Additional structure should be defined within the CMS templates:

Content Management

Within `setting.py` we can define so called **Templates** which are than available over django CMS toolbars **Page > Templates** UI. These templates can have a different structures. In the boilerplate there are four predefined templates:

- fullsize.html
- sidebar_left.html
- sidebar_right.html
- tpl_home.html

When choosing a name, be descriptive about their uses as the customer can set them by himself. If I would add a more narrow header option for fullsize, I would simply call it `fullsize_simple.html`.

Menu

All menu relevant templates are kept within `templates/includes/menu/*.html`. These display what classes are used to render a navigation, breadcrumb or even the pagenav.

Messages

You need to be aware of the django message framework which displays global notifications or error messages. This file is kept within `templates/includes/messages.html` and included within `templates/base.html`.

Analytics

Store all analytics code within the designated file in `templates/includes/analytics.html` which will be injected right after the opening `<body>` tag. Google Analytics is already pre-prepared and will be shown when adding the required UA-XXXXX code within the CMS.

1.2.3 Reference

We use a combination of various frontend libraries to create a fast and robust boilerplate.

Normalize

- http://necolas.github.io/normalize.css/

Bootstrap

- http://getbootstrap.com

SASS/SCSS

- http://sass-lang.com/
- http://compass-style.org/

1.3 Stylesheets

This chapter describes in more detail what the **css** guidelines are, how they are structured and what the requirements are:

1.3.1 Guidelines

- Use **4 space indentation** and **not** tabs
- Use **underscores** for scss file naming
- Use **double-quotes** `"` for all text values
- Use **dashes** to separate class/id names, **not** camelCase or underscore
- Do **not** overuse nesting! If you got only one instance, use one line
- Always add a space after the colon
- Only write one css property per line
- Keep `sass/layout/` clean and use the available structure
- Use `sass/sites/` for theme based or specific styles
- Define settings within `sass/settings/`
- Avoid referencing css using their parent like div.container
- Use shorthands for values like `#ccc` or `white`
- Use full words instead of shorthands like `number` instead of `nr`
- Avoid using universal selectors for maintainability/performance reasons
- Use progressive enhancement whenever possible
- Validation is not required but nice

Style

Use block-style and group elements underneath:

1. includes (compass includes)
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)

Combine attributes such as background-image, background-color, background-repeat into `background: #fff url("image.png") no-repeat left top;`.

Also ensure combined css selectors are always on a new line.

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;
    }
}
```

Nesting

With great power comes great responsibility (just wanted to throw that in here). When writing in **sass** or **less** we sometimes forget performance over laziness. While nesting is very powerful, we should avoid unnecessary levels or blocks that can be achieved simpler. A good example is the following code:

```
.nav-main {
    ul {
        li {
            a {
                color: red;
            }
        }
    }
}
```

This can be optimised in various ways. First of all, we don't need the additional nesting. When no other styles are needed just simply write compact: `.nav-main ul li a { color:  red; }`

Another optimisation is to think about the required declaration levels. Do we really need the *ul li* to declare our anchor red? Can it just simply be `.nav-main a { color:  red }`?

When we are using multiple styles, we might even consider a structure such as:

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

Which makes our code more structured and readable.

1.3.2 Structure

Every folder within `private/sass/` includes a file called **_all.scss**. This file is included within `private/sass/base.scss` which gets compiled into `static/css/base.css`. Update the all file to include additional modules, do not include files directly within *base.scss*.

The SCSS file is structured into 3 separate section for **mobile**, **tablet** and **desktop**. This allows for an easy responsive approach by maintaining the code within a single file.

Hint: The first line `// @media all` is commented in order to use the **sass** `@extend` functionality which is currently not available within *@media* rules.

addons/

Separate modules which are plug-n-play able and add them into this folder. Traditionally these are django addons which can be installed such as **aldryn-blog**, **aldryn-news** or **aldryn-shop**.

layout/

Layout specific styles such as header, footer or general forms should be added here. Also specific definitions for print, retina or mobile only styles that are used globally should be defined here.

In addition you can set fonts, icons and custom mixins.

libs/

As foundation, we use normalize.css as many other boilerplates are using in its default state.

We are using the foundation grid with **24 columns** and a max-width of **960px** which offers the most flexible way of designing and a readable code.

We include various helper classes inspired by *bootstrap* within `private/sass/libs/_bootstrap.scss`. It makes sense to read the code as most elements are setup using the settings within `private/sass/settings/default.scss`.

These files should generally **not** be overwritten.

settings/

Control over color, sizes and other settings are found here. These settings have mostly impact on the available libraries. You can add additional settings file if required. For example `private/sass/settings/_shop.scss`.

sites/

If you are working on a theme-based setup or have styles which do not fit into the folders described above, this is the appropriate place to add them.

This folder can be freely structured. `_custom.scss` can be used for quick fixes or hacks.

Hint: **Deep Nesting** It can often happen that you end up with large sites files like `_marketing.scss` and `_application.scss`. In order to modularise those files and create a better overview, you can create an additional folder and include all required files within the original scss files. This could end up with a structure as illustrated underneath.

```
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)
```

1.3.3 Reference

You can use the full power of the Django template language. Additionally the following libraries are on your disposal:

Django CMS

In order for Django CMS to work, you need to include the **css** and **js** sekizai blocks and add `{% cms_toolbar %}` after the closing `</body>` tag.

Django Sekizai

With sekizai you can include additional assets such as CSS or JavaScript. Simply add `{% load sekizai_tags %}` on top of your file and use `{% addtoblock "js" %}` or `{% addtoblock "css" %}`.

When including a single file, do not add any white spaces or breaks inside. Sekizai validates code for dublicates and comfortably only includes one instance. So if you already include jQuery, Sekizai will only render it **once**.

The output is rendered within `{% render_block "css" %}` and `{% render_block "js" %}` in `templates/base_root.html`.

Example

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

Django Compress

Django compressor should also be enabled within your setup. This allows you to compress files automatically on a live system.

Example

```
{% load compress %}

{% compress js %}
<script src="{% static "js/base.js" %}"></script>
<script>obj.value = 'value';</script>
{% endcompress %}
```

Aldryn Snake

Aldryn snakes behaves similar to django-sekizai but is mostly used within the backend. The output is rendered within `{{ ALDRYN_SNAKE.render_head }}` and `{{ ALDRYN_SNAKE.render_tail }}`.

Aldryn snakes allows the additional insertion of html fragments or any other textual data.

1.3.4 Mistakes

There are several mistakes I find from time to time over again which I would like to clarify:

Floating

When using `float: left;`, `display: block;` is not required anymore as **every** element which is floated is automatically a **block** element.

Hidden

With modern HTML5 we can use the html attribute‘‘hidden=”hidden”‘‘ which is a **softer** `display: none;` and can easy be overwritten using css or JavaScript. This attribute is ideal for hiding elements which should be later displayed using JavaScript, as there is no delay in which the element is hidden as for typical dynamic execution.

1.4 Images

1.4.1 Optimization

Images are the number one source of optimisation when it comes to file size. Optimise images using tools like CodeKit or Grunt.

1.5 JavaScript

This chapter describes in more detail what the **JavaScript** guidelines are, how they are structured and what the requirements are:

1.5.1 Guidelines

- Use **4 space indentation** and **not** tabs
- Use camelCase for variables and **not** underscores or dashes
- Use **dot** annotation `.` for javascript file naming
- Use **single-quotes** `'` for **all** values
- Use a maximum length of 120 characters, however 80 is preferred
- Use `base.js` for global and general functions and avoid adding js files to the root
- Use the frameworks prefix inside the `addons` folder
- Use the module and singleton pattern to structure code
- Use the `js-` prefix when working with JS related selectors and do not add stylings to it
- JavaScript should validate JS Lint
- Use full words instead of shorthands like `number` instead of `nr`
- Keep <script> and the following starting enclosure on the same level
- Separate all script tags within a `{% addtoblock "js" %}`
- Do not use inline JS within HTML attributes such as `onclick=""` or `onload=""`
- Do not use inline JS within HTML, try to implement JavaScript files only
- Do not add spaces when writing `if (true) {} else {}` or `function helloWorld() {}`
- Always use semicolons and full brackets except shortcuts like `var i = (true) ? 'yes' : 'no';` or single lines `if (index <= 0) index = 0;`
- Always declare variables on top of the functions and not in-between
- Never use $ for variable names like `var $el = $('.el');`
- Never use comma separation for variable declerations like `var a, b, c;`
- Ensure that JavaScript widgets don't create disturbances while the DOM is loading
- Please make sure that & has a character reference like "&amp;"

Additionally follow the "Code Conventions for the JavaScript Programming Language": http://javascript.crockford.com/code.html

Example

```
<script>
jQuery(document).ready(function () {

    var Cl.MyApp = {
        load: function {
            alert('hello world');
        }
    };

    // load application
    Cl.MyApp.load();

});
</script>
```

Prefixing

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* define styles according to BEM principles and *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 ok.

1.5.2 Structure

We use `static/js/base.js` as a single point of entry. Within this file, we lazy load additional elements using require.js.

In addition, we use `Cl` as global namespace for all our custom code or addons to keep the global namespace clean. Following this guideline, we use as initiation Cl.Base or Cl.Application for our website. See `static/js/addons/cl.utils.js` as example.

Always add the appropriate prefix to the filename. If there are multiple libraries used within the file, the wrapping namespace should win. In case of Cl.Utils we use jQuery, MBP and class.js. Cl wins as Cl.Utils is the wrapper.

addons/

Separate modules which are plug-n-play able and add them into this folder. Traditionally the classjs-plugins are added here.

If you use external addons, such as jquery.select2, ensure that those files get added minified. Try to avoid changing the code of external addons as this over complicates the update process if hotfixes are released.

libs/

We include three major frameworks to help us with JavaScript: jQuery and class.js

class.js is simpler than the bloated jQuery UI and offers faster performance. It simply provides a more traditional class based inheritance model but still uses prototypal inheritance.

Additional libraries such as respond.js or html5.js provide html/css shivs for older browsers.

tests/

Using QUnit you can create your JavaScript Unit tests here. This is a very simple setup and the folder structure within this area can be customized according to your needs.

1.5.3 Reference

jQuery

- http://jquery.com

class.js

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

Shivs

- https://github.com/aFarkas/html5shiv
- https://github.com/scottjehl/Respond
- https://code.google.com/p/swfobject/
- http://patik.com/blog/complete-cross-browser-console-log/

Contribution

2.1 Contribution

You are very welcome to help improving the aldryn-standard-boilerplate, especially the documentation. Feel free to fork and send us pull requests.

To extend and run the documentation, you will have to manually install Sphinx. The automated setup takes care of the rest:

1. navigate to docs `cd docs`
2. run `make init` to install additional requirements
3. run `make run` to let the server run

When opening localhost:8000 the screen might appear blank. This is due to the fact that the docs/_build folder is not yet created. Simply change something within an *.rst file and refresh the page. Livereload will than take care of the rest to auto refresh your page on change.