ARISE

Frontend Development Guidelines

Introduction

These guidelines are designed to help those involved in developing frontend code for our projects understand the standards required, keep consistency between projects and work efficiently.

The core elements of the frontend stack we use and you, should be very familiar with, consists of...

- HTML5
- SASS
- JQuery
- Neat 4

It may also help to have familiarity with some **additional elements** we often use...

- Smarty
- Gulp
- JQuery UI

And **for Laravel projects only**, you may also need experience with webpack and Blade templates.

HTML Structure

In almost all cases, pages consist of a <header>, multiple <section> tags and a <footer>.

The <section> tags will essentially become template blocks which can be stacked up and populated to make new pages via the CMS.

SASS & CSS

We use a very simplified version of the BEM model to keep CSS readable, consistent and reusable. All class names will be lowercase and hyphenated if required.

Parent classes

Each of the <section> variations we mentioned get their own SCSS file and parent class, e.g. `.template-name`

Child classes

Any elements within the parent class then receive a child class, extending the parent class name with two underscores followed by the sub-element's own name.

For example, sections will often include a container (a fixed-width central column) as a child element, for example...

```
<section class=”template-name”>
     <div class=”template-name__container”>
     </div>
</section>
```

You can use SASS’s & operator to organise the CSS for this neatly...

```
.template-name {
     &__container {
     }
}
```

Variant classes

Sometimes there will be multiple variants of a single template. Rather than duplicating the code, we use variant classes to extend the parent class, this time adding two hyphens and a descriptor.

```
<section class=”template-name template-name--jazzy-background”>

</section>
```

Again, you can use SASS’s & operator to organise the CSS for this neatly...

```
.template-name {
    &--jazzy-background {
    }
}
```

Containers

As mentioned, we often use a container as a fixed-width limited column within content. A default container is usually defined with a max-width set, and then we use SASS’s extend function to apply this.

```
.container {
    width: 100%;
    max-width: 1160px;
    margin: 0 auto;
    padding: 0 20px;
}
```

```
.template-name {
    &__container {
        @extend .container;
    }
}
```

Grid system

We use the Neat 4 grid system with a standard 12 column grid with 20-pixel gutters.

```
.template-name {
    &__half {
        @include grid-column(6);
    }
}
```

Directory structure

The SCSS code will generally follow this structure...

- *assets*
  - *fonts*
  - *images*
  - *js*
  - *scss*
    - *base*
      - *_base.scss*
      - *_type.scss*

- *components*
  - *<_component-name>.scss*
- *objects*
  - *<_object-name>.scss*
- *sections*
  - *<_section-name>.scss*
- *settings*
  - *_colours.css*
  - *_fonts.scss*
- *tools*
  - *_custom.scss*
- *main.scss*

base/_base.scss
This file contains the basic layout code that will usually apply to all projects, for example, your .container class will live here.

base/_type.scss
The basic typography classes will live here.

sections/
The sections subdirectory will include an individual file for each <section> template you created, with its name reflecting the parent class, e.g. .template-name sits within _template-name.scss.

objects/

Similarly to the sections subdirectory, this will include multiple files each with a name matching the class within. Objects are essentially any repeatable element that is not a <section> template, for example, buttons or form input styles may live in here.

components/
Similarly to the sections and objects subdirectories, this will include multiple files each with a name matching the class within. Components are essentially anything that's not a section template or repeatable object, for example, the header and footer code often live here.

settings/_colours.css
Colour variables that will be used throughout the rest of the SCSS will be stored here.

settings/_fonts.scss
Font variables and imports that will be used throughout the rest of the SCSS will be stored here.

tools/_custom.scss
Any custom mixins are stored here.

main.scss
This brings it all together by importing files from each directory, as well as any files loaded externally or via a package manager. When creating a new file don't forget to add it here!

Javascript

We use the JQuery library for any Javascript required and occasionally use this for layout adjustments or interactions.

Directory structure

- *assets*
    - *fonts*
    - *images*
    - *js*
        - *includes*
            - *<filename>.js*
        - *<filename>.js*
        - *layout.js*
    - *scss*

includes/
Any files that should be loaded before the others should be stored in this directory.
layout.js
Includes a few common functions that we use to make layout adjustments.

<filename>.js
Any other code is stored in a file with a relevant name, e.g. code that is to do with navigation could be saved within navigation.js.

All files within the js directory will be included when Gulp is run. Any Javascript loaded via package manager is included within the Gulp file.

Class names

To maintain separation between appearance and interaction, no layout CSS class names should be referenced in the Javascript. Instead, add an additional class preceded with .js-, for example…

```
<button class=”button button--highlight js-send-form”>
     Send Message
</section>
```

Images

Vector graphics

Where possible we use vector graphics to keep things clear, crisp and quick. These are usually in the form of SVG files and are included in an <img> tag or as a background image.

Backgrounds

While we have largely relied on CSS’s background-size: cover for hero images and fancy backgrounds, this method has performance and SEO shortfalls, instead we recommend using a combination of display:flex, object-fit: cover, and the <picture> tag.

```
<section class=”hero”>
```

```
    <picture class="hero__picture">
        <img src="assets/images/filename.jpg"
             alt="Caption text"
             loading="lazy"/>
    </picture>
</section>
```

And the SCSS for this would look something like...

```
.hero{
  &__picture{
    height: 100%;
    width: 100%;
    display: flex;
```

```
    img {
      width: 100%;
      height: auto;
      object-fit: cover;
    }
  }
}
```

Responsive

While some of the responsive requirements will be shown in provided designs, we tend to adapt most of this in the browser. Firstly, we create the desktop view of the site and then adapt code by creating breakpoints where things break on resize, using media queries.

For the most part, columns will reduce as the screen size decreases. SCSS also allows us to place media queries within the relevant files and classes as appropriate.

```
.template-name {
```

```
&__image {
     @include grid-column(3);
}
&__text {
     @include grid-column(9);
}

@media (max-width: 1102px)
{
     &__image {
          @include grid-column(1);
     }
     &__text {
          @include grid-column(11);
     }
}
```

```
        @media (max-width: 604px)
        {
            &__image {
                @include grid-column(12);
            }
            &__text {
                @include grid-column(12);
            }
        }
}
```

Smarty

The Smarty template language is used to separate code into reusable files (e.g. a separate header.tpl file), and keep logic and structure somewhat separate. Deep knowledge of Smarty is not necessary, as you can populate templates using plain HTML code. While providing the finished front-end templates in Smarty format is preferable, if your workflow is quicker sticking with flat HTML files, that's fine too.

Front-end Framework Tool

We've put together a helpful framework including the basic file structure and Smarty template previews. This may also be helpful as a practical example of some of the concepts explored above.

You will find the latest version of this framework along with its own documentation at
https://github.com/benjwalker/arise-frontend-framework

Freedom & improvements

These guidelines are largely for your guidance and to explain the way we currently do things. They are a product of experience and the result of input from many lovely designers and developers we've worked with over the years. Please feel free to also contribute to that evolution if you can see potential improvements to the way we do things