BEM Architecture & Pre-Processor

Aamer Qanadilo

CONTENTS

What is BEM?

- BEM was created by Yandex, the "Russian Google" in 2009.
- It is a component based architecture and naming convention. It stand for Block / Element / Modifier.

Why do we use it?

One of the most common critic for BEM is that it is ugly, and that those long classes names are making the markup hard to read.It's very sad because the true beauty of BEM is hidden.

THE GOOD STUFFS ABOUT BEM

- you can know which classes are related to others
- you can also know in which file they are declared, reducing headache
- you can understand the role of the class (And therefore better follow the SRP)

SRP: Single Responsibility Principle

THE BENIFITS OF USING BEM

- get self documented code, and logically reduce the need for CSS documentation
- keep specificity low (because most of your selectors will only be composed of one classe)
- make class colision almost impossible (all classes will be uniques)
- making it safer to have multiple sources of code (any CSS frameworks mixed with your own)

Guidelines for using Blocks

- The block name describes its purpose ("What is it?" — menu or button), not its state ("What does it look like?" — red or big).

```
<!-- Correct. The `error` block is semantically meaningful -->
<div class="error"></div>

<!-- Incorrect. It describes the appearance -->
<div class="red-text"></div>
```

- Blocks can be nested in each other.

```
<body>
  <!-- `header` block -->
  <header class="header">
      <!-- Nested `logo` block -->
      <div class="logo"></div>

      <!-- Nested `search-form` block -->
      <form class="search-form"></form>
  </header>
</body>
```

Guidelines for using Elements

- The Element name describes its purpose ("What is it?" — item or text), not its state ("What does it look like?" — red or big).
- The structure of an element's full name is block-name__element-name. The element name is separated from the block name with a double underscore (__).

```
<!-- `search-form` block -->
<form class="search-form">
    <!-- `input` element in the `search-form` block -->
    <input class="search-form__input">

    <!-- `button` element in the `search-form` block -->
    <button class="search-form__button">Search</button>
</form>
```

Guidelines for using Elements cont.

- An element is always part of a block, not another element. This means that element names can't define a hierarchy such as block__elem1__elem2.

```
<!--
    Correct. The structure of the full element name follows the pattern:
    `block-name__element-name`
-->
<form class="search-form">
    <div class="search-form__content">
        <input class="search-form__input">

        <button class="search-form__button">Search</button>
    </div>
</form>
```

```
<!--
    Incorrect. The structure of the full element name doesn't follow the pattern:
    `block-name__element-name`
-->
<form class="search-form">
    <div class="search-form__content">
        <!-- Recommended: `search-form__input` or `search-form__content-input` -->
        <input class="search-form__content__input">

        <!-- Recommended: `search-form__button` or `search-form__content-button` -->
        <button class="search-form__content__button">Search</button>
    </div>
</form>
```

Should I create a Block or an Element

- **Create a Block** If a section of code might be reused and it doesn't depend on other page components being implemented.

- **Create an Element** If a section of code can't be used separately without the parent entity (the block).

Guidelines for using Modifier

- The modifier name describes its appearance ("What size?" or "Which theme?" and so on — size_s or theme_islands), its state ("How is it different from the others?" — disabled, focused, etc.) and its behavior ("How does it behave?" or "How does it respond to the user?" — such as directions_left-top).

- The modifier name is separated from the block or element name by a single underscore (_).

Types of Modifier

- **Boolean,** The structure of the modifier's full name follows the pattern:
    1. block-name_modifier-name
    2. block-name__element-name_modifier-name

```
<!-- The `search-form` block has the `focused` Boolean modifier -->
<form class="search-form search-form_focused">
    <input class="search-form__input">

    <!-- The `button` element has the `disabled` Boolean modifier -->
    <button class="search-form__button search-form__button_disabled">Search</button>
</form>
```

Types of Modifier cont.

- **Key-value,** The structure of the modifier's full name follows the pattern:
    1. block-name_modifier-name_modifier-value
    2. block-name__element-name_modifier-name_modifier-value

```
<!-- The `search-form` block has the `theme` modifier with the value `islands` -->
<form class="search-form search-form_theme_islands">
    <input class="search-form__input">

    <!-- The `button` element has the `size` modifier with the value `m` -->
    <button class="search-form__button search-form__button_size_m">Search</button>
</form>
```

Modifiers can’ t be used alone

- From the BEM perspective, a modifier can’t be used in isolation from the modified block or element. A modifier should change the appearance, behavior, or state of the entity, not replace it.

```
<!--
  Correct. The `search-form` block has the `theme` modifier with
  the value `islands`
-->
<form class="search-form search-form_theme_islands">
    <input class="search-form__input">

    <button class="search-form__button">Search</button>
</form>
```

```
<!-- Incorrect. The modified class `search-form` is missing -->
<form class="search-form_theme_islands">
    <input class="search-form__input">

    <button class="search-form__button">Search</button>
</form>
```

Mixing the BEM entities

- Combine the behavior and styles of multiple entities without duplicating code.

- Create semantically new UI components based on existing ones.

```
<!-- `header` block -->
<div class="header">
    <!--
        The `search-form` block is mixed with the `search-form` element
        from the `header` block
    -->
    <div class="search-form header__search-form"></div>
</div>
```

File Structure

- A single block corresponds to a single directory.
- The block and the directory have the same name. For example, the header block is in the header/ directory, and the menu block is in the menu/ directory.
- A block's implementation is divided into separate technology files. For example, header.css and header.js.
- The block directory is the root directory for the subdirectories of its elements and modifiers.

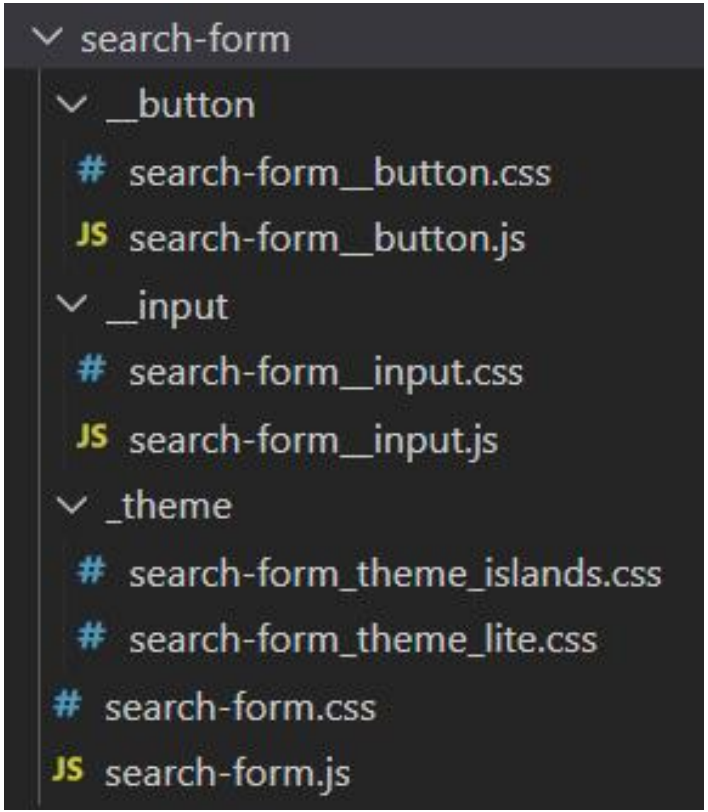
File Structure cont.

- Names of element directories begin with a double underscore (__). For example, header/__logo/ and menu/__item/.

- Names of modifier directories begin with a single underscore (_). For example, header/_fixed/ and menu/_theme_islands/.

- Implementations of elements and modifiers are divided into separate technology files. For example, header__input.js and header_theme_islands.css.

File Structure cont.



What are CSS Pre-Processors?

- You may be thinking, okay, CSS sounds great, what could go wrong? There are still some downsides. As the web gets more advanced, writing regular vanilla (plain) CSS can get **lengthy** and **repetitive**.

- **CSS preprocessors extend the functionality of regular CSS.** They add more logical syntax and tools like **variables**, **if/else statements**, and **loops**. This makes the CSS more efficient and concise, powerful and dynamic. Using a CSS preprocessor, a developer is able to write out more complex style and layout. The source code can be shorter and more readable.

How does it work?

- To accomplish this goal, CSS preprocessors add syntax that is not within CSS itself. More advanced CSS is written that extends the basic functionalities. This advanced code is later compiled as normal CSS code that the browser can understand.

Pre-Processor Variables

- Just like in regular programming languages, CSS preprocessors give you the opportunity to add variables to your styles. This is helpful for styles you plan to reuse often.

```
$hoverColor: #33FF9B;

button {
  background-color: $hoverColor;
}
```

Don’t we have variables in CSS?

- It’s true that we can declare variables in CSS and we can even modify them using JavaScript or through the CSS Code.

```
:root {
  --color: red;
}

body {
  background-color: var(--color);
}

document.documentElement.style.setProperty('--color',
'blue');
```

- But we can’t ignore the fact that we can’t use these variables with the if/else statement that the Pre-Processors have provided for us!

Pre-Processor if/else statement

- in Sass preprocessor you can use the @if statement to create conditional styles based on variables. Here is an example:

```
$color: red;

@if $color == red {
  background-color: red;
} @else if $color == blue {
  background-color: blue;
} @else {
  background-color: green;
}
```

Pre-Processor Loops

- Loops are useful when you have a collection of items (arrays or objects) that you want to increment over.
- As an example, let's say we have an object of our different social media icons and the colors they should be. We want to look through and apply the relevant color and link to each button.

```
$social: (
  'facebook': #4267B2,
  'twitter': #1D9BF0 ,
  'linkedin': #0072b1,
  'instagram': #8a3ab9,
);
```

```
@each $name, $color in $social {
  // selector based on href name
  [href*='#{$name}'] {
    background: $color;

 // apply the link for the relevant image file
 &::before {
      content: url(https://www.careerfoundry.com/images/#{$name}.png);
    }
  }
}
```

Pros of using a CSS preprocessor

- **It makes your code more maintainable.** For example, you can declare your brand colors in one place: $primaryColor, $secondaryColor, etc. If your brand colors change later, you only have to update them in one place now.

- **Write DRY CSS, a.k.a. Don't Repeat Yourself.** CSS preprocessors make it easy to reuse styles, meaning you don't have to write the same code over and over.

- **They make your code more organized.** Rather than sprawling sheets of styles, you can group your code and be more specific. Less repetition is shorter and more readable.

Cons of using a CSS preprocessor

- **Debugging is harder.** Since you're reusing code, it could take longer to find where the problem is.

- **Additional complication time.** Since the browser doesn't read this more advanced version of CSS, it needs to compile it into regular CSS before showing the style.

- **Can produce very large CSS files.** The source files will be more concise, but the generated CSS files could be huge. This could cause additional time for a request to complete.

Popular CSS preprocessors

- There are three main CSS preprocessors. SASS, LESS, and Stylus. Most CSS preprocessors have similar features. Yet each one has its own unique way of completing the same task.

- All three preprocessors allow you to create variables, media queries, mixins, nesting, loops, conditionals, and import. There are some differences when it comes to advanced usage.

BEM & Pre-Processors

- BEM can make it easier to use preprocessors like Sass or Less by allowing you to nest selectors and create variables for commonly used values. This can help you write more maintainable and reusable code

Classic CSS

```css
#opinions_box h1 {
    margin: 0 0 8px 0;
    text-align: center;
}

#opinions_box {
    p.more_pp {
        a {
            text-decoration: underline;
        }
    }

    input[type="text"] {
        border: 1px solid #ccc!important;
    }
}
```

BEM

```scss
.opinions_box {
    margin: 0 0 8px 0;
    text-align: center;

    &__view-more {
        text-decoration: underline;
    }

    &__text-input {
        border: 1px solid #ccc;
    }

    &--is-inactive {
        color: gray;
    }
}
```

T H A N K S

Aamer Qanadilo

Foothill Technology Solutions