BEM 101

Robin Rendle on Apr 2, 2015 (Updated on Mar 9, 2016)

The following is a collaborative post by guest Joe Richardson (http://lifes.gd/) , Robin Rendle, and a bunch of the CSS-Tricks staff. Joe wanted to do a post about BEM, which we loved, and just about everybody around here had thoughts and opinions about BEM, so we figured we'd all get together on it and do it together.

The **Block, Element, Modifier** methodology (commonly referred to as BEM (https://en.bem.info/method)) is a popular *naming convention* for classes in HTML and CSS. Developed by (https://en.bem.info/) the team at Yandex, its goal is to help developers better understand the relationship between the HTML and CSS in a given project.

Here's an example of what a CSS developer writing in the BEM style might write:

```css
/* Block component */
.btn {}

/* Element that depends upon the block */
.btn__price {}

/* Modifier that changes the style of the block */
.btn--orange {}
.btn--big {}
```

In this CSS methodology a **block** is a top-level abstraction of a new component, for example a button: `.btn { }`. This block should be thought of as a parent. Child items, or **elements**, can be placed inside and these are denoted by two underscores following the name of the block like `.btn__price { }`. Finally, **modifiers** can manipulate the block so that we can theme or style that particular component without inflicting changes on a completely unrelated module. This is done by appending two hyphens to the name of the block just like `btn--orange`.

The markup might then look like this:

```html
<a class="btn btn--big btn--orange" href="https://css-tricks.com">
  <span class="btn__price">$9.99</span>
```

```
    <span class="btn__text">Subscribe</span>
  </a>
```

If another developer wrote this markup, and we weren't familiar with the CSS, we should still have a good idea of which classes are responsible for what and how they depend on one another. Developers can then build their own components and modify the existing block to their heart's content. Without writing much CSS, developers are potentially capable of creating many different combinations of buttons simply by changing a class in the markup:

Embedded Pen Here

At first this syntax might seem slower than simply making a new class for each type of button, but this is not the case for several reasons we'll cover.

(#aa-why-should-we-consider-bem) Why should we consider BEM?

1. If we want to make a new style of a component, we can easily see which modifiers and children already exist. We might even realize we don't need to write any CSS in the first place because there is a pre-existing modifier that does what we need.
2. If we are reading the markup instead of CSS, we should be able to quickly get an idea of which element depends on another (in the previous example we can see that `.btn__price` depends on `.btn`, even if we don't know what that does just yet.)
3. Designers and developers can consistently name components for easier communication between team members. In other words, BEM gives everyone on a project a declarative syntax that they can share so that they're on the same page.

Harry Roberts identified (http://csswizardry.com/2015/03/more-transparent-ui-code-with-namespaces) another key benefit of using a syntax like BEM when he writes about improving developer confidence:

> *This is the main reason we end up with bloated code bases, full of legacy and unknown CSS that we daren't touch. We lack the confidence to be able to work with and modify existing styles because we fear the consequences of CSS' globally operating and leaky nature. Almost all problems with CSS at scale boil down to confidence (or lack thereof): People don't know what things do any more. People daren't make changes because they don't know how far reaching the effects will be.*

Likewise, Philip Walton argues (http://philipwalton.com/articles/side-effects-in-css/) that this problem can be fixed if enough developers stick to the principles of BEM:

> *While 100% predictable code may never be possible, it's important to understand the trade-offs you make with the conventions you choose. If you follow strict BEM conventions, you will be able to update and add to your CSS in the future with the full confidence that your changes will not have side effects.*

So if developers can work on a project more confidently, then they're sure to make smarter decisions about how these visual components should be used. This methodology might not be a perfect cure for all these ailments, but it certainly gives developers a standard on which to write better, more maintainable code in the future.

Another smart part of BEM is that **everything is a class** and **nothing is nested**. That makes CSS specificity very flat and low, which is a good idea (https://css-tricks.com/strategies-keeping-css-specificity-low/) . It means you won't end up fighting with yourself over specificity.

Let's take a look at some of the problems with BEM...

(#aa-problems-with-bem-css) Problems with ~~BEM~~ CSS

Of course nobody will twist your arm if you break from BEM rules. You could still write a CSS selector like this:

CSS
```css
.nav .nav__listItem .btn--orange {
  background-color: green;
}
```

That looks like it has parts of BEM going on, but it's not BEM. It has nested selectors, and the modifier doesn't even accurately describe what's going on. If we did this, we'd be screwing up the specificity flatness that is so helpful with BEM.

A block (such as `.nav`) should never override the styles of another block or modifier (such as `.btn--orange`). Otherwise this would make it almost impossible to read the HTML and understand what this component does; in the process we're bound to greatly shake another developer's confidence in the codebase. This goes for HTML, as well: what would you expect if you saw the following markup?

HTML
```html
<a class="btn" href="https://css-tricks.com">
  <div class="nav__listItem">Item one</div>
  <div class="nav__listItem">Item two</div>
</a>
```

What's probably going on here is that an element in a completely unrelated block has the code a developer needed, but the child elements don't require a `.nav` class as the parent. This makes for an exceptionally confusing and inconsistent codebase which should be avoided at all costs. So we can summarize these problems by:

1. Never overriding modifiers in an unrelated block.
2. Avoiding making unnecessary parent elements when the child can exist quite happily by itself.

(#aa-more-examples-of-bem-in-action) More examples of BEM in action

(#aa-accordion-demo) Accordion demo

Embedded Pen Here

In this example there is one block, two elements and one modifier. Here we've can created an `.accordion__copy-open` modifier which lets us know we shouldn't use it on another block or element.

(#aa-navigation-demo) Navigation demo

Embedded Pen Here

This navigation demo has 1 block, 6 elements and 1 modifier. It's even perfectly OK to create blocks without modifiers at all. At some point in the future a developer can always bolt on (or bind to) new modifiers so long as the block remains consistent.

(#aa-dislikes-of-bem) Dislikes of BEM

Perhaps you don't like the double-underscores or double-dashes thing. Fine, use something else that is unique that you will consistently enforce.

Here's another sentiment:

Not found

> *Not sure I'm sold on BEM. .site-search .site-search__field .site-search–full Why not: .site-search .site-search input .site-search .full*

— Samuel Fine (@samuelfine) March 11, 2015 (https://twitter.com/samuelfine/status/575645771334291456)

Those *last* three selectors all have different specificity levels. They either require parents or not. Without any rules in place, they don't *say* as much as the ones on top.

Is it possible that this tiny, isolated example feels perfectly fine to you and never ends up biting you in the butt? Perhaps. But the more CSS you have in a project, the more little things like this add up, the more specificity and complexity battles you go through.

Not found

> *BEM sounds super useful if you don't know how HTML or CSS work.*
>
> *— Samuel Fine (@samuelfine) March 11, 2015 (https://twitter.com/samuelfine/status/575646836251344897)*

Not to pick on Samuel here, but his sentiments are shared by a lot of people so it makes for a good example. They see BEM, and they just outright reject it. If you want to dislike BEM, that's absolutely fine, but I think it would be hard to argue that **having a set of rules** that aid in understanding and assist in keeping CSS maintainable is a bad idea.

In the SMACSS methodology, you're likely to find a CSS classname with three letters. Modifiers then follow the module name with a hyphen:

```css
/* Example Module */
.btn { }

/* Modifier of the btn class */
.btn-primary { }

/* Btn Module with State */
.btn.is-collapsed { }
```

That's just a different naming approach to the same kind of problem. It's pretty similar, but you're just being more specific about dependencies and keeping specificity flatter.

In OOCSS (http://oocss.org/) , blocks are similarly generic.

css
```
/* Example Module */
.mod { }

/* Part of Module */
.inner { }

/* Talk Module */
.talk { }

/* Variation of part inside Module */
.talk .inner { }
```

So you would use multiple classes in the HTML for variations. The inside part isn't named like it has a dependency, so it is less clear but potentially more reusable. BEM would do `.mod__inner` and `.mod--talk` and `.mod--talk__inner`.

These are just variations on methodology. Remember that nobody is twisting your arm here, these are self-imposed rules where the value comes from following them.

(#aa-sass-and-bem) Sass and BEM

For those of you writing Sass and enjoy nesting as a way of scoping styles, you can still author in a nested format, but get CSS that isn't nested, with `@at-root`:

scss
```
.block {
  @at-root #{&}__element {
  }
  @at-root #{&}--modifier {
  }
}
```

Gets you:

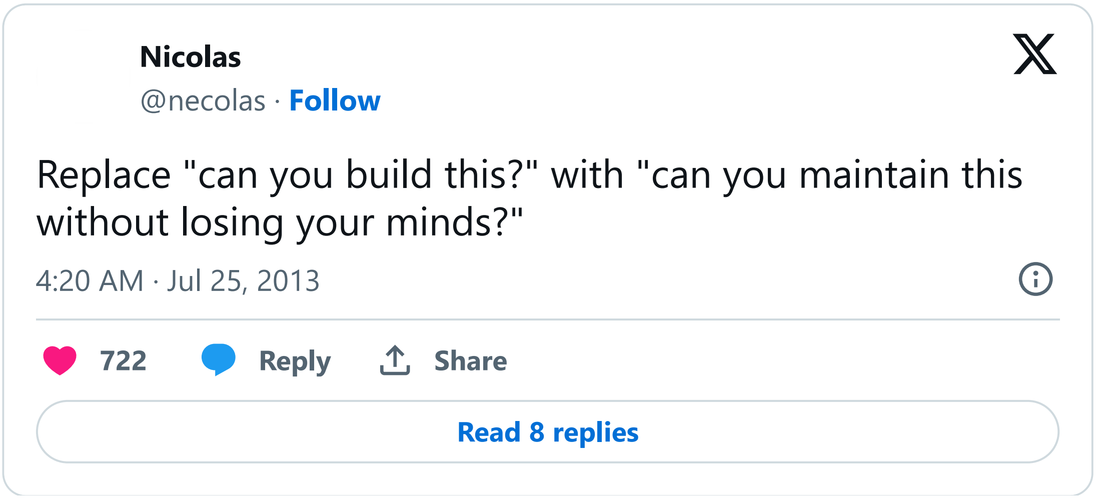
css
```
.block {
}
.block__element {
}
.block--modifier {
}
```

And you can get as abstract as you want! Check out Danield Guillan's BEM Constructor (https://github.com/danielguillan/bem-constructor) or Anders Schmidt Hansen's Expressive BEM (http://codepen.io/andersschmidt/post/expressive-bem-with-sass-a-different-approach) .

(#aa-summary) Summary

To wrap things up I think it's fair to say that even though BEM won't solve all our problems it is extraordinarily useful for constructing scalable and maintainable interfaces where everyone on the team should have a clear idea of how things can be improved. This is because a great deal of front end development is not just about the nice tricks that solve one little problem in the short term; we need agreements, promises and binding social contracts between developers so that our codebase can adapt over time.

Generally I like to think of BEM as an answer to Nicolas Gallagher's question:



Nicolas
@necolas · Follow

Replace "can you build this?" with "can you maintain this without losing your minds?"

4:20 AM · Jul 25, 2013

722 Reply Share

Read 8 replies

(#aa-further-reading) Further reading

- The BEM project website (https://en.bem.info/)
- Side effects in CSS (http://philipwalton.com/articles/side-effects-in-css/)
- Chaining BEM modifiers (http://webuild.envato.com/blog/chainable-bem-modifiers/)

- BEM principles (http://www.smashingmagazine.com/2012/04/16/a-new-front-end-methodology-bem/)
- More transparent UI code with namespaces (http://csswizardry.com/2015/03/more-transparent-ui-code-with-namespaces)