Scalable and Modular Architecture for CSS

A flexible guide to developing sites small and large.

by Jonathan Snook

SMACSS: Scalable and Modular Architecture for CSS
http://smacss.com

ISBN 978-0-9856321-0-6

Snook.ca Web Development, Inc.
Ottawa, Ontario, Canada
http://snook.ca

Second edition

Categorizing CSS Rules

Every project needs some organization. Throwing every new style you create onto the end of a single file would make finding things more difficult and would be very confusing for anybody else working on the project. Of course, you likely have some organization in place already. Hopefully, what you read among these pages will highlight what works with your existing process and, if I'm lucky, you will see new ways in which you can improve your process.

How do you decide whether to use ID selectors, or class selectors, or any number of selectors that are at your disposal? How do you decide which elements should get the styling magic you wish to bestow upon it? How do you make it easy to understand how your site and your styles are organized?

At the very core of SMACSS is categorization. By categorizing CSS rules, we begin to see patterns and can define better practices around each of these patterns.

There are five types of categories:

1. Base
2. Layout
3. Module
4. State
5. Theme

We often find ourselves mixing styles across each of these categories. If we are more aware of what we are trying to style, we can avoid the complexity that comes from intertwining these rules.

Each category has certain guidelines that apply to it. This somewhat succinct separation allows us to ask ourselves questions during the development process. How are we going to code things and *why* are we going to code them that way?

Much of the purpose of categorizing things is to codify patterns—things that repeat themselves within our design. Repetition results in less code, easier maintenance, and greater consistency in the user experience. These are all wins. Exceptions to the rule can be advantageous but they should be justified.

Base rules are the defaults. They are almost exclusively single element selectors but it could include attribute selectors, pseudo-class selectors, child selectors or sibling selectors. Essentially, a base style says that wherever this element is on the page, it should look like *this*.

Examples of Base Styles

```
html, body, form { margin: 0; padding: 0; }
input[type=text] { border: 1px solid #999; }
a { color: #039; }
a:hover { color: #03C; }
```

Layout rules divide the page into sections. Layouts hold one or more modules together.

Modules are the reusable, modular parts of our design. They are the callouts, the sidebar sections, the product lists and so on.

State rules are ways to describe how our modules or layouts will look when in a particular state. Is it hidden or expanded? Is it active or inactive? They are about describing how a module or layout looks on screens that are smaller or bigger. They are also about describing how a module might look in different views like the home page or the inside page.

Finally, **Theme rules** are similar to state rules in that they describe how modules or layouts might look. Most sites don't require a layer of theming but it is good to be aware of it.

Naming Rules

By separating rules into the five categories, naming convention is beneficial for immediately understanding which category a particular style belongs to and its role within the overall scope of the page. On large projects, it is more likely to have styles broken up across multiple files. In these cases, naming convention also makes it easier to find which file a style belongs to.

I like to use a prefix to differentiate between Layout, State, and Module rules. For Layout, I use `l-` but `layout-` would work just as well. Using prefixes like `grid-` also provide enough clarity to separate layout styles from other styles. For State rules, I like `is-` as in `is-hidden` or `is-collapsed`. This helps describe things in a very readable way.

Modules are going to be the bulk of any project. As a result, having every module start with a prefix like `.module-` would be needlessly verbose. Modules just use the name of the module itself.

Example classes

```
/* Example Module */
.example { }

/* Callout Module */
.callout { }

/* Callout Module with State */
.callout.is-collapsed { }

/* Form field module */
.field { }

/* Inline layout  */
.l-inline { }
```

Related elements within a module use the base name as a prefix. On this site, code examples use `.exm` and the captions use `.exm-caption`. I can instantly look at the caption class and understand that it is related to the code examples and where I can find the styles for that.

Modules that are a variation on another module should also use the base module name as a prefix. Sub-classing is covered in more detail in the Module Rules chapter.

This naming convention will be used throughout these pages. Like most other things that I have outlined here, don't feel like you have to stick to these guidelines rigidly. Have a convention, document it, and stick to it.