Extracted from:

Modern CSS with Tailwind

Flexible Styling Without the Fuss

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic exPress

Modern CSS with Tailwind

Flexible Styling Without the Fuss

Noel Rappin

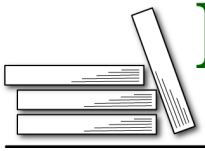
edited by Katharine Dvorak

Modern CSS with Tailwind

Flexible Styling Without the Fuss

Noel Rappin

The Pragmatic Bookshelf
Raleigh, North Carolina



For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Katharine Dvorak
Copy Editor: Corina Lebegioara
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-818-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—May 2021

Tailwind seems like a counter-intuitive solution to the problem of managing CSS for a complex site. Tailwind is made up of many, many small utility CSS class names, most of which set one specific CSS property to one specific value. The preferred way to get complex behavior in Tailwind is to compose multiple CSS classes together on the HTML element.

This goes against a lot of CSS naming conventions that have been developed over the years. Many CSS frameworks and naming conventions suggest using names that reflect the semantic meaning of the element on the page—names like `button`, `nav-bar`, or `menu-item`.

Tailwind classes aren't semantic at all. They're all utility classes, meaning a Tailwind class represents a specific element on a page, like `font-bold` for text formatting or `m-6` for margin. Many CSS frameworks include utility classes but consider the semantic class names more important. Using Tailwind and utility classes suggests a lot of duplication, as Tailwind utility classes are often repeated on multiple DOM elements.

Despite the potential duplication, Tailwind can work on larger sites.

When you apply a Tailwind class at any particular point, both the nature of the styling change and the scope of that change are exceptionally clear. Tailwind's short names may seem cryptic at first, but the naming patterns are strong and become easier to read. Also, Tailwind's prefixes make it easy to define special behavior in the HTML, such as hover and responsive behaviors on differently sized screens. The prefixes also make what's going on clearer from just the HTML.

Because the Tailwind classes are endlessly combinable, you write far less external CSS code in Tailwind than you might in another CSS style. You don't need to name as many custom CSS classes when using Tailwind. And because the Tailwind changes are so closely tied to the HTML markup, it's easier to predict the result of making a change.

Tailwind allows you to extract a own CSS class from a list of Tailwind utilities and to give it a more semantic name. Rather than create your own classes, Tailwind recommends taking advantage of the same tools you use in your front-end stack to reduce duplication. For example, rather than create a separate CSS class for button styles, Tailwind suggests that you create a reusable React component or a Rails partial or helper method, and you define the CSS styles only once for that reusable item.

Tailwind is made up of a few different pieces: the utility classes that we'll spend the bulk of our time working with in this book, a reset stylesheet, and functions that make working with Tailwind easier.

Utilities

Tailwind's utility classes are the most important part of Tailwind to understand. Here's how they work and how I'll talk about them in the book.

Tailwind is made up of hundreds and hundreds of utility classes, most of which set the value of a single CSS property. For example, the .font-bold Tailwind utility class is an alias for the CSS property, font-weight: 700. You'd use that utility in an HTML element as part of the class attribute, as in class=font-bold.

The Tailwind system generates a set of utility CSS classes based on the Tailwind configuration file, and you have a great deal of freedom over which classes are included and even the names Tailwind uses. Unless I clearly say otherwise, in this book I'll be talking about the default set of classes generated by a minimal configuration, and in Chapter 8, Customizing Tailwind, on page ?, I'll talk about how to adjust which classes are available.

The Leading Dot

When you see Tailwind utilities outside their context in an HTML element, sometimes they may be written with a leading dot to indicate that they're referring to a CSS class, as you saw earlier with .font-bold. When they're used in HTML, though, the leading dot isn't used. In this book, I'll use the leading dot to indicate a Tailwind utility.

Tailwind utilities often come in families with a common pattern of beginnings or endings. When I talk about those, I'll use syntax like this: .text-{size}, to indicate a family of utilities that include .text-xs, .text-sm, .text-xl, and so on. When this syntax is used, the dash is only needed if the part in braces is there, so you'll see text-sm but also text.

The variable part of the utility name doesn't have to be at the end of the name. For example, in margin sizing utilities, .m{direction}-{size} indicates a family of utilities such as .m-0 or .mt-10. As you'll see, the variable part of utilities is often consistent across different parts of Tailwind. For example, the options for {size} and {direction} in the margin utilities are shared by the padding utilities and several other utility families.

Preflight

When you install Tailwind, you need to import three different files with the commands: `@tailwind base`, `@tailwind components`, and `@tailwind utilities`. Each of these files contains a different set of CSS rules. (In the webpack-based installation, you import with `@import` as the command name rather than `@tailwind`.)

`@tailwind base` contains Tailwind's reset stylesheet called *Preflight*. A reset stylesheet is a restyling of all the base HTML elements to a minimal set of styling properties. Without a reset stylesheet, each browser defines its own default set of style properties for how to render individual HTML elements that don't have further CSS properties. Using a reset stylesheet gives our application control over this baseline, eliminating differences between browsers and providing a more minimal backdrop into which we insert our own custom styling.

You can see the full set of reset styles Tailwind uses by looking at the file, `node_modules/tailwindcss/dist/base.css`. Essentially, though, Preflight does a few things:

- It overrides all styling from headers, so, say, an `h1` is visually identical to the base text.
- It removes styling from `ul` and `ol` lists, resulting in no bullets by default, which is an ironic thing to mention in a bulleted list.
- It sets all margins to zero for elements that would normally have margins.
- It sets all borders to a 0-pixel width, solid, and the defined border color by default.
- It sets images and image-like objects to `display: block` rather than `display: inline`, meaning they'll set themselves up as separate paragraphs (as if they were `div` tags), rather than inline (as if they were `span` tags).

If you only use the Preflight styles, you'll get a pretty boring page. But that's the point. Using Preflight ensures that any change to the display properties are affirmatively and explicitly added by us.

The `@tailwind components` file is quite small, and it only consists of the definitions of the `container` CSS classes, which are usually used at the top level of a page to define the box that the whole page is drawn in. I'll talk about this more in Chapter 5, Page Layout, on page ?.

The bulk of what's considered to be Tailwind is in the `@tailwind utilities` file, which defines all the utilities and their prefixed variants. I'll spend most of this book describing the contents of this file.