sitepoint

TAILWIND CSS:
CRAFT BEAUTIFUL, FLEXIBLE, AND RESPONSIVE DESIGNS

BY IVAYLO GERCHEV

THE RAPID WAY TO DEVELOP CSS

Tailwind CSS: Craft Beautiful, Flexible, and Responsive Designs

Ebook ISBN: 978-1-925836-51-6

- **Author:** Ivaylo Gerchev
- **Series Editor:** Oliver Lindberg
- **Product Manager:** Simon Mackie
- **Technical Editor:** Shahed Nasser
- **English Editor:** Ralph Mason
- **Cover Designer:** Alex Walker

Notice of Rights

Notice of Liability

the authors and SitePoint Pty. Ltd., nor its dealers or distributors will be held liable for any damages to be caused either directly or indirectly by the instructions contained in this book, or by the software or hardware products described herein.

Trademark Notice

Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.

Published by SitePoint Pty. Ltd.

10-12 Gwynne St, Richmond, VIC, 3121
Australia
Web: www.sitepoint.com
Email: books@sitepoint.com

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit http://www.sitepoint.com/ to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, design, and more.

About the Author

Ivaylo Gerchev is a web developer/designer from Bulgaria. In his free time he likes to write articles and tutorials sharing his knowledge and understanding on various web development topics. His favorite topics include UI, UX, SVG, HTML, CSS, Tailwind, JavaScript, Node, Nest, Adonis, Vue, React, Angular, PHP, Laravel, and Statamic. The best tools he uses are Figma and VS Code. When he's not programming the Web, he loves to program his own reality.

Preface

Conventions Used

Code Samples

Code in this book is displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park.
The birds were singing and the kids were all bac
```

You'll notice that we've used certain layout styles throughout this book to signify different types of information. Look out for the following items.

Tips, Notes, and Warnings

HEY, YOU!

Tips provide helpful little pointers.

AHEM, EXCUSE ME ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

MAKE SURE YOU ALWAYS ...

... pay attention to these important points.

WATCH OUT!

Warnings highlight any gotchas that are likely to trip you up along the way.

Supplementary Materials

- https://www.sitepoint.com/community/ are SitePoint's forums, for help on any tricky problems.
- **books@sitepoint.com** is our email address, should you need to contact us to report a problem, or for any other reason.

Getting Started with Tailwind CSS

There are two main types of CSS framework. One is based around *components*—a group that includes frameworks such as Bootstrap, Foundation, and Bulma. The other type of CSS framework is based around *utilities*—a group that includes the likes of Tachyons, Tailwind CSS, and Windi CSS.

COMPONENT VS UTILITY CLASSES

If you're not clear on the difference between component and utility classes, jump to the "What Is a Utility Class?" section below, and then continue reading from here.

For many years, component-based frameworks were the de facto standard for building websites quickly and easily. But all this magic comes with a price. Without serious customization, sites built with such frameworks look similar to each other. And customization is a real pain in the neck for anyone who wants to build something more complex and/or creative. Component-based styles are easy to implement, but inflexible and confined to certain boundaries. Solving specificity issues while trying to override the default styles of a particular framework isn't a fun and productive job.

“Utility-first” frameworks were created to solve this problem. A **utility-first** framework is built with low-level functionality in mind. Utility classes offer much more power and flexibility than component classes.

Utility-first frameworks provide the following advantages:

- Utility classes operate at a low level. This means we have more control and flexibility over how we apply them—a concept that’s similar to the power and flexibility offered by a low-level language like C or C++, in contrast to high-level languages such as JavaScript or PHP.
- Utility classes are easy to customize, so we can build any design.
- A utility-first approach scales well. It’s great for managing and maintaining large projects, because we only have to maintain HTML files, instead of a large CSS codebase. It’s already used with success by big sites like GitHub, Heroku, Kickstarter, Twitch, and Segment.
- Utility classes can be adopted to any design.
- Utility classes are completely customizable and extensible. It’s easier to build unique, custom website designs without fighting with unwanted styles.
- Utility classes allow for much easier implementation of responsive design patterns.

- Utility classes have consistent styles, which gives us a ready-to-use design system. We can also create our own design system if we need to.
- With utility classes, we can still extract common, repetitive patterns into custom, reusable components. But in contrast to predefined components, custom components will be exactly what we need.

In summary, we can say that a utility-first framework gives us balance between the concrete and the abstract.

Now that we've seen how useful utility-first frameworks can be, it's time to pick one and see what it can do for us in action. In this book, we'll explore Tailwind CSS, which is the most popular of the utility-first frameworks.

What Is Tailwind?

Tailwind is a set of low-level, reusable utility classes that can be used like building blocks to create virtually any design we can imagine. This utility-first framework covers the most important CSS properties, but it can be easily extended in a variety of ways. It can be used either for rapid prototyping or for creating full-blown designs.

Tailwind has great documentation, covering every class utility in detail and showing the ways it can be customized. There's also a play-

ground where you can try out your ideas. You can also check out Tailwind's screencasts if you prefer to learn by watching.

As of version 2.0, Tailwind CSS supports the latest stable versions of Chrome, Firefox, Edge, and Safari. There's no support for any version of IE, including IE11.

What Is a Utility Class?

As we already know, Tailwind's main characteristic is the use of utility classes. But what is a utility class?

While a **component class** is a *collection* of predefined CSS settings applied in an opinionated fashion, a **utility class** is mostly a *single* CSS property or behavior that we can use freely in a predictable way. This gives us the freedom to combine, mix and match different settings depending on our requirements. We have greater control over each element's appearance. We can change and fine-tune an element's appearance much more effortlessly with utility classes.

In Bootstrap, we create a button by using predefined component classes, as in the following example:

```
<button class="btn btn-success">Success</button>
```

Here, the `btn` and `btn-success` are the so-called component classes. Each one of them represents a collection of predefined CSS settings.

In Tailwind, we create a button by using utility classes:

```
<button class="py-2 px-4 bg-green-500 text-white
  Success
</button>
```

Here, `py-2`, `px-4`, `bg-green-500`, and so on, are the so-called utility classes, and each one of them represents just a single CSS setting. To create the button, we use multiple utilities—which we put together like the pieces of a puzzle in order to get the whole picture.

This CodePen demo shows these two buttons on the same page.

The buttons looks quite similar, but they're created in different ways.

We've covered the basic difference between component and utility classes, but let's now take an even closer look at utility classes.

In Tailwind, the CSS `font-style: italic` is represented by the `italic` utility class.

Here are some more examples:

- `text-align: right` becomes `text-right`
- `font-weight: 700` becomes `font-bold`
- `border-radius: 0.25rem` becomes `rounded`
- `width: 0.5rem` becomes `w-2`
- `padding: 1.5rem` becomes `p-6`

And here's how classes are applied in practice. Let's say we want to make a paragraph bold and italic. We do it this way in CSS:

```
p {
  font-weight: 700;
  font-style: italic;
}
```

To do the same in Tailwind, we add the `font-bold` and `italic` classes directly to the HTML element:

```
<p class="font-bold italic">Lorem ipsum...</p>
```

In Tailwind, we can also apply classes based on an element's states, such as hover, focus, and so on. For example, if we want the above paragraph to be italic only on mouse hover, we can write the class like this: `hover:italic`.

As you can see, Tailwind utility classes are mostly self-explanatory. We can often imagine how the styled element looks by just reading the classes. Some class names are heavily abbreviated, admittedly, but once we've grasped the pattern and had a bit of practice with them, they're easy to remember and recall.

What a Design System Is, and How It Can Help Us

As Tailwind offers a sort of design system, it will be useful to say few words about what a design system is and how it can facilitate the design process.

In simple terms, a **design system** is a set of rules and conventions for how a design should be built. It includes predefined rules about sizes, colors, text, and so on. Traditionally, when we build a design we need to make multiple choices about things like:

- the size of the design elements (text, images, etc.)
- the colors and color variations
- the fonts and other typographic features

… and so on.

Making a decision for every small part of a design can lead to choice paralysis and inconsistency. It's tedious and error-prone. It would be much easier if we first established a design system with already pre-

defined options that are tested and proven to work. We can then just select from the existing, limited set of options and combine them to produce the desired outcome.

This is actually what Tailwind gives us—a well-crafted design system that we can use to speed up, smooth, and facilitate our design-building process.

Up and Running with Tailwind

KNOW YOUR HTML AND CSS

To follow along with the rest of this tutorial, you should have a good understanding of HTML and CSS and their concepts. If you're not up to speed with those yet, check out SitePoint's HTML and CSS learning path.

The official documentation describes a bunch of different ways to install Tailwind. In this tutorial, we'll use the simplest one—which involves including Tailwind in our projects via the Play CDN option. So just create an HTML file and put the following content in it:

```
<!doctype html>
<html>
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-wi
```

```
    <link href="https://cdnjs.cloudflare.com/ajax/l
    <script src="https://cdn.tailwindcss.com"></scr
  </head>
  <body>

    <!-- ... -->
  </body>
  </html>
```

This is the starting template we'll build upon throughout the rest of the tutorial. We'll also include a Font Awesome link that we'll use for the icons in our design.

Exploring Tailwind Basics

There are four main factors involved in every web design project:

- **Layout**. It all starts with a blueprint. This defines how the white-space and elements of our design are organized and ordered.
- **Typography**. This includes all text content, including messages.
- **Colors**. This brings life to a design and defines a design's mood and brand.
- **Imagery**. This includes the visuals of a design, such as icons, images, illustrations, and so on.

In the next four sections, we'll learn more about each one of these factors and see how Tailwind can help us to implement them in the development phase. The aim here is to give you a bird's-eye view of what classes to look for when you're working on a particular component. Don't worry: I'll go into much more detail for each class when we start exploring a practical example later on in the chapter.

Responsive Web Design

Our coverage of layout, typography, color and imagery here won't be able to include principles and techniques relating to responsive web design, as that topic is beyond the scope of this book. But it's an important topic that's central to modern web design. For more information, see this general overview of responsive web design, and also Tailwind's documentation for specific instructions.

Layout

In this section, we'll explore briefly the most commonly used classes for layout composition. We can group classes by their function, as follows:

- **Size**. This includes width and height utilities for setting an element's dimensions.
- **Space**. This includes margin and padding utilities for adding space in our design.

- **Position**. This includes an element's positioning and coordinates.
- **Borders**. This includes various utilities for setting an element's borders, such as style, width, and radius.
- **Display**. This includes the way elements are rendered.

In modern CSS, we have also Flexbox and Grid classes for building a layout. We'll cover only the Flexbox utilities in this chapter, as they're much simpler and easier to use for beginners.

When we use Flexbox, we start by creating a flex container by adding a `flex` class to a parent element. Then we add additional flex classes to configure how the flex items inside the container (direct children) will be displayed. For example, to render flex items vertically as a column, we add a `flex-col` class along with the `flex` class:

```
<div class="flex flex-col">
  <div>1</div>
  <div>2</div>
  <div>3</div>
</div>
```

This is the base for applying flex classes. There are plenty of them in Tailwind, and it would be tedious to explain each one individually. Instead, when we explore a practical example later on, I'll cover the flex classes we use in that example.

Typography

Now that we have a layout, the next step is to add the content we want to display. This is mostly done by using text. Here are the most commonly used text utilities grouped by function:

- **Font**. This includes font family, size, style and weight utilities, as well as tracking and leading settings.
- **Text**. This includes text aligning, color and opacity, decoration and transformation.
- **List**. This includes list type and position styling.

Colors

We have a layout, and we have text. Now we need to bring life to them by using some colors. Tailwind offers a large, pre-made color palette. Applying a color is super easy. Here are the two most common uses of color:

- **Text**. To apply a color to text we use the pattern `text-[color]-[number]`. The `number` variable defines tints and shades. For example, to make text dark red, we can use a `text-red-900` class. To make it light red, we can use `text-red-100`.
- **Background**. To use a color as a background, we use the pattern `bg-[color]-[number]`.

Imagery: Icons and Images

The final spice in our design recipe is the visual imagery. Visuals are like salt and spices: a meal isn't tasty without them. The most commonly used visuals are:

- **Icons**. These can be based on SVGs or icon fonts. As we saw earlier, we included Font Awesome in our template. To use an icon from, it we use the pattern `fas fa-[icon-name]`. For example, to use a search icon for a search input, we can use the `fas fa-search` classes. Notice that `fas` placed before the icon name means that we use Font Awesome's solid icons collection, which is free. Font Awesome offers some base styling for its icons, but we can style them with Tailwind's utilities (for color, size, etc.) as well.
- **Images**. To style images, we can use a bunch of Tailwind classes, such as width and height, opacity, shadows, borders, filters, and so on.

Building a Blog Starter Template

In this last section, we'll explore how to build a simple blog starter template from scratch. I won't go too deeply into the detail of each individual utility class, but I'll provide enough explanation where it's needed.

Here's the final version of the blog template. You can also test it in action in this CodePen demo.

UTILITY CLASS HELP

For information about any particular utility class, you can use Jay Elaraj's handy Tailwind cheatsheet, or you can search for a specific class in the Tailwind documentation.

BASE STYLES

As we dive into building our starter template with Tailwind, it's important to note that Tailwind applies an opinionated set of base styles for every project by default.

Creating the Header

We'll build the template from top to bottom, starting with a header. The following image shows what we're trying to create.

To create the header, put the following code inside the `<body>` element in the starting template:

```
<div class="container mx-auto">
  <header class="flex justify-between items-cente
    <div class="flex-shrink-0 ml-6 cursor-pointe
      <i class="fas fa-wind fa-2x text-yellow-50
      <span class="text-3xl font-semibold text-bl
    </div>
    <ul class="flex mr-10 font-semibold">
      <li class="mr-6 p-1 border-b-2 border-yello
        <a class="cursor-default text-blue-200" h
      </li>
      <li class="mr-6 p-1">
        <a class="text-white hover:text-blue-300"
      </li>
      <li class="mr-6 p-1">
        <a class="text-white hover:text-blue-300"
      </li>
      <li class="mr-6 p-1">
        <a class="text-white hover:text-blue-300"
      </li>
    </ul>
  </header>
</div>
```

Let's break the header's code into smaller blocks. First, we've wrapped all the content in a container by adding the `container` class in the wrapping `<div>` element:

```
<div class="container mx-auto">

</div>
```

This forces the design to accommodate certain dimensions depending on the current breakpoint. We've also centered the design with the `mx-auto` utility. This sets the left and right margins to `auto`.

In Tailwind, when `x` is used after a CSS setting abbreviation (`m` for margin here), it means that the setting will be applied both on *left* and *right*. If `y` is used instead, it means the setting will be applied both *top* and *bottom.*

The reason we create such a container is that, on large screens, the design will be centered and presented in a more compact size, which in my opinion looks much better than a fully-fluid viewport.

The next thing we've done is create the header with a `<header>` element:

```
<header class="flex justify-between items-center

</header>
```

We've created a flex container and used `justify-between` and `items-center` classes to add an equal amount of space between flex items and align them along the center of the container's cross axis.

We've also used the `sticky` and `top-0` classes to make the header fixed to the top when users scroll down, and we've set a `z-10` class to ensure the header will be always on top.

We've added a shade of blue color as a background and some padding for both top and bottom sides of the header.

The first item in the header's container is the blog's logo:

```
<div class="flex-shrink-0 ml-6 cursor-pointer">
  <i class="fas fa-wind fa-2x text-yellow-500"></
  <span class="text-3xl font-semibold text-blue-2
</div>
```

It's combination of a yellow colored wind icon (`fas fa-wind`) and light blue colored "Tailwind School" text. We've made the icon bigger by using Font Awesome's `fa-2x` class. The text is made bigger and semibold by using Tailwind's `text-3xl` and `font-semibold` classes respectively.

For the logo's container, we've added a bit of left margin and used the `flex-shrink-0` class to prevent the logo from shrinking when the design is resized to smaller viewports.

The second item in the header's container is the main navigation:

```
<ul class="flex overflow-x-hidden mr-10 font-sem

</ul>
```

We've created it by using a `ul` element turned into a flex container so we can style its items as horizontal links. We've used the `over-flow-x-hidden` class to clip the content within navigation that overflows its left and right bounds. We've also added some right margin.

The `mr-10` class and the `ml-6` (logo) classes use the `r` for *right* and `l` for *left* abbreviations to set right and left margin respectively. In a similar way, `t` and `b` can be used for setting *top* and *bottom* sides of an element.

For the navigation's links, we've added some right margin and a small padding to all sides:

```
<li class="mr-6 p-1 border-b-2 border-yellow-500"
  <a class="cursor-default text-blue-200" href="
```

```
    <a class="cursor-default text-blue-200" href="#
  </li>

  <li class="mr-6 p-1">
    <a class="text-white hover:text-blue-300" href=
  </li>
  <li class="mr-6 p-1">
    <a class="text-white hover:text-blue-300" href=
  </li>
  <li class="mr-6 p-1">
    <a class="text-white hover:text-blue-300" href=
  </li>
```

When we use a setting like padding without side abbreviation (`p-1` here), it's applied to all sides.

We've set the color of links to white, which changes to light blue on hovering. We've also used the `hover:` prefix to achieve that effect.

We've styled the active "Home" link by adding a thin yellow border below the text. The border is created with the `border-b-2` class, where `b` is for *bottom* and `2` is the amount of border thickness.

THE DESIGN PROCESS

In a text-based tutorial, it's hard to demonstrate how a design is built step by step, and how each step is built upon the previous one. For that reason, I suggest you

watch this short video, which shows an example of the steps and reasoning involved in the design process. It should give you some extra insight into the process we're following here.

Chapter 2: Going Beyond the Basics

As we learned from the chapter, Tailwind CSS is the best known of the utility-first frameworks. It offers a rich collection of CSS class utilities that can be combined like Lego blocks to build any kind of design. Learning the basics of how to use these utilities is also very straightforward.

But this knowledge alone isn't enough for building complex and flexible designs. After grasping the basics, instead of diving deeper into what can be done with Tailwind, many users decide to use ready-made templates or a copy-paste approach to building their designs. The aim of this chapter is to avoid such a scenario by providing some insights into Tailwind's more advanced capabilities. We'll dive a little deeper into what we can do with Tailwind, learning how to create reusable, utility-based components and templates, and how to make our designs responsive. We'll also start to explore how Tailwind can be configured and customized—a topic we'll continue to explore and build on in the following chapters.

All these skills will help us create a more flexible and manageable codebase. They'll let us go beyond the Tailwind basics so we can build our own components and templates with confidence.

Getting Started with Tailwind

In this section, we'll learn how to set up a new project with Tailwind.

PROJECT CODE

You can find the finished project for this chapter in the code repo for this book.

The project creation process described below will also be used as a starting point in the following parts of the book. In each of the next parts, you'll be redirected to this section to prepare a new project.

GETTING READY

To follow along from here onwards, you'll firstly need to have Node installed on your machine.

Also, make sure you have a basic knowledge of Tailwind's utilities—including how they work and how they're applied to HTML—as we won't be explaining them in detail here. The basics of Tailwind utilities were covered in the first chapter.

UTILITY CLASS REFERENCES

For information about particular utility classes, you can use Jay Elaraj's handy Tailwind cheatsheet, or you can search for a specific class in the Tailwind documentation.

Base Styles

Remember to bear in mind that Tailwind applies an opinionated set of base styles for every project by default.

The first step is to create a new Node app. In a directory of your choice, run this command:

```
npm init -y
```

In the chapter, we played with Tailwind by using a Play CDN link. That's a great way to get started when we're doing a quick test or proof of concept experiment. But in this chapter, we'll use the Tailwind CLI, which will give us a production-ready setup.

The first step is to install Tailwind:

```
npm install -D tailwindcss
```

Next, run the following command:

```
npx tailwindcss init
```

NPX

> `npx` is a tool that's automatically installed alongside `npm` and allows you to run a command from a local or remote npm package.

This will create a minimal `tailwind.config.js` file, where we can put our customization options during development. The generated file contains the following:

```
// /tailwind.config.js
module.exports = {
  content: [],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

The next thing to do is to add our template paths in the content section, so Tailwind can compile the utilities used in our templates. In our case, we'll add just an `index.html` file, which we'll create a bit later:

```
// /tailwind.config.js
module.exports = {
  content: ["index.html"],
```

```
    ...
}
```

The next step is to create a `styles.css` file, in the root directory, where we'll include the framework's styles using the @tailwind directive:

```
/* /styles.css */
@tailwind base;
@tailwind components;
@tailwind utilities;
```

As the names suggest, Tailwind styles are divided into three groups. The first group represents all the base styles—such as CSS resets. The second represents the styles for components—where any custom components are also injected. The third group contains all of Tailwind's default utilities, as well as any custom-made utilities.

The next step is to build the Tailwind styles. To do this, run the following command:

```
npx tailwindcss -i styles.css -o tailwind.css
```

This gets the styles from the `styles.css` file as an input and generates a `tailwind.css` file as an output.

If we don't want to run this command every time we make some changes, we can append the `-w` or `--watch` flag at the end, which will rebuild our styles every time we make a change:

```
npx tailwindcss -i styles.css -o tailwind.css -w
```

To facilitate the use of both commands, let's define them as scripts in `package.json` file:

```
// /package.json
...
"scripts": {
  "dev": "npx tailwindcss -i styles.css -o tailwi
  "dev:watch": "npx tailwindcss -i styles.css -o


},
...
```

Now we're ready to start playing with Tailwind.

Creating Tailwind Components

As you may have expected, no matter how useful Tailwind classes are, we'll soon realize that we have repeating groups of utilities in our

code. Of course, this makes the code error-prone and hard to maintain. This is where components come into play.

Components are predefined sets of utilities that can be reused and adapted for various scenarios. A component effectively allows us to reuse Tailwind's classes, which reduces the code repetition and improves maintainability.

Tailwind allows us to extract classes into reusable components in two major ways. The first way is to just extract them as groups of classes. The second way is to create a reusable component with a frontend framework like Vue or React, or a template partial with a template engine like Twig or Blade. We'll explore both scenarios in the following sections.

But before we dive into the examples, let's create an `index.html` file in the root directory and add links to the generated `tailwind.css` file and to the Font Awesome icons. Here's what the starter template should look like:

```
<!-- /index.html -->
<!doctype html>
<html>
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-w
    <link rel="stylesheet" type="text/css" href="
```

```
      <link rel="stylesheet" type="text/css" href=
      <link rel="stylesheet" href="https://cdnjs.cl
    </head>
    <body>
      <!-- ... -->
    </body>
  </html>
```

Extracting Classes into Reusable Components

Extracting classes into components is as easy as grouping a series of classes by using the @apply directive. To demonstrate how this can be done, we'll create an alert component. First, let's see how our alert component looks when it's built only with utility classes:

```
<div class="flex flex-col p-4 pt-2 w-full border-
  <div class="font-semibold italic text-lg text-b
  <div class="leading-tight text-sm text-blue-80(
</div>
```

Here's a CodePen demo of the code above.

As you can see, this code sample creates a simple "Info" alert component. But the problem is that, if we want to create different alerts—such as "Warning", "Success", and so on—we'll need to repeat a big

part of the code for each alert. This leads to poor maintainability, because if we wanted to change the overall styles of these alerts, we'd need to update each one separately. To avoid such a scenario, we'll extract the repeating patterns into individual component classes.

Open the `styles.css` file and add the following code:

```
/* /styles.css */
...

@layer components {
  .alert {
    @apply flex flex-col p-4 pt-2 w-full border-l
  }

  .alert-title {
    @apply font-semibold italic text-lg;
  }

  .alert-content {
    @apply leading-tight text-sm;
  }

  .alert-info {
    @apply bg-blue-100 border-blue-500
  }

  .alert-info-title {
```

```
    .alert-info-title {
      @apply text-blue-500;
    }

    .alert-info-content {
      @apply text-blue-800;
    }

    .alert-warning {
      @apply bg-red-100 border-red-500
    }

    .alert-warning-title {
      @apply text-red-500;
    }

    .alert-warning-content {
      @apply text-red-800;
    }
  }
```

Here, we've used the `@layer components { ... }` directive to wrap our custom component classes. This is to tell Tailwind which layer those styles belong to and to avoid specificity issues. The options are `base`, `components`, and `utilities`.

First, we extract the base code for each alert component part into an individual class (`.alert` , `.alert-title` , and `.alert-content`). Then, we extract the code that differs for each individual alert (the specific color classes). For example, for the “Info” alert, the classes would be `.alert-info` , `.alert-info-title` , and `.alert-info-content` . Generally, we extract or group the utilities into smaller and more manageable component classes.

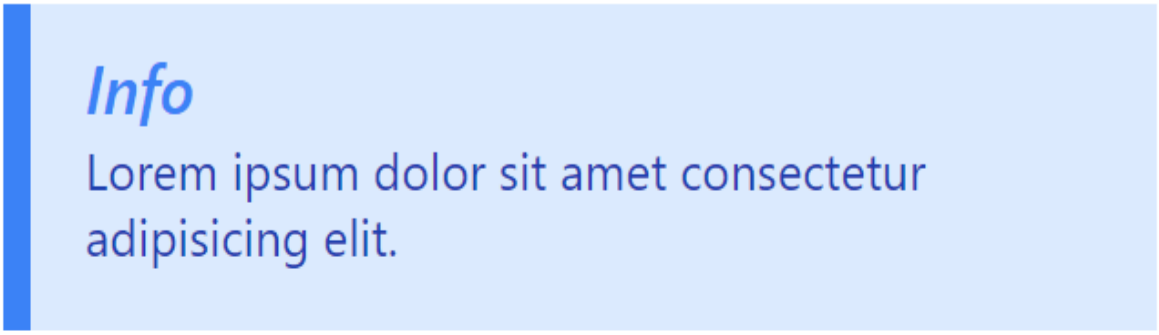
As you might have noticed, the classes we’ve just created are pretty similar to those in component-based frameworks such as Bootstrap or Bulma. The advantage of Tailwind’s classes is that they’re more transparent and easier to tweak. We can see exactly which utilities are applied and we can easily edit them whenever we need to.

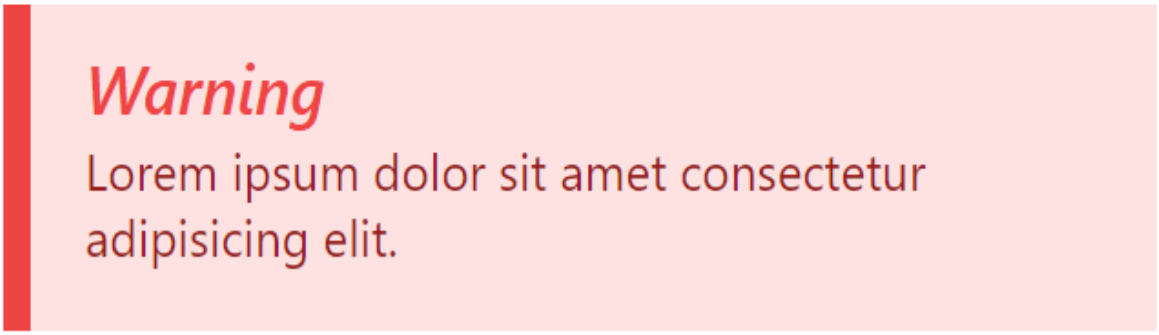
Once we’ve created the required classes, we need to build Tailwind styles again (by running `npm run dev` or `npm run dev:watch`) for the changes to take effect. Once that’s done, we can try out the new components in action. Open the `index.html` file and add the following code:

```
<!-- /index.html -->
<div class="alert alert-info m-6 w-1/3">
  <div class="alert-title alert-info-title">Info<
  <div class="alert-content alert-info-content">I
</div>
<div class="alert alert-warning m-6 w-1/3">
```

```
    <div class="alert-title alert-warning-title">Wa
    <div class="alert-content alert-warning-content
  </div>
```

This code creates “Info” and “Warning” alerts.





As you can see, with our custom component classes we can create much more manageable and maintainable code. We now have fewer classes to use and fewer places to make changes.

Building Tailwind Components with Vue

Extracting component classes, as demonstrated above, works well for fairly small, simple components. If we want to create a complex component with multiple elements, we'll need to use a more flexible approach. By "flexible" I mean more reusable and easier to manage. As we've seen already, even for our simple alert component, we needed to create many component classes for the different states or parts of the component. Imagine if we wanted to create a fairly complex component such as a card. Doing this by using the `@apply` directive would be quite impractical. After all, we want to simplify our code, not to make it a mess, right?

Another way we can utilize the full potential of Tailwind is by using it in frontend frameworks (like Vue, React, Svelte) that allow us to create reusable components. This will give us much more flexibility when we want to build complex components. We'll demonstrate this by creating a general recipe card component with Vue.

Pizza Margherita

Invented in Naples in honor of the first queen of Italy, the Margherita pizza is the triumph of Italian cuisine in the world.

1h 15m | 4 Servings | Easy

LEARNING VUE

If you're not familiar with Vue, you can consult the Vue documentation, or take a look at *Jump Start Vue.js*, which offers a

more in-depth exploration of Vue's features and abilities.

For this example, we'll need to add a link to the Vue framework:

```
<!-- /index.html -->
<script src="https://unpkg.com/vue@3"></script>
```

Let's first see what the recipe card component's code would look like without Vue:

```
<div class="m-5 shadow-md w-80 rounded overflow-h
  <img src="https://source.unsplash.com/x00CzBt4I

  <div class="p-2">
    <div class="font-bold text-lg text-gray-700 l
      <a href="#" class="hover:underline">Pizza N
    </div>
    <p class="text-xs leading-tight tracking-wide
    <div class="flex pt-2 border-t border-gray-30
      <div class="flex-1 text-center">
        <i class="far fa-clock"></i>
        <p>1h 15m</p>
      </div>
      <div class="flex-1 text-center">
        <i class="fas fa-utensils"></i>
        <p>4 Servings</p>
```

```
        </div>
        <div class="flex-1 text-center">
          <i class="fas fa-signal"></i>
          <p>Easy</p>
        </div>
      </div>
    </div>
  </div>
```

Here's a CodePen demo of the code above.

The problem here is that, if we want to use this template for multiple recipes, we'll end up with lots of code repetition. Later on, if we decide to change the card's design, we'll need to edit the template in multiple places. This is error-prone and wasteful of our time. The solution is to find a way to extract the repeating code into a reusable template. We can easily do this by creating a reusable Vue component.

Open `index.html` and add the following code before the closing `</body>` tag:

```
// /index.html
<script>
const app = Vue.createApp({})

app.component('recipe-card', {
```

```
  props: ['imageurl', 'imgalt', 'titleurl', 'tit
  template: `
    <div class="m-5 shadow-md w-80 rounded overfl
      <img :src="imageurl" :alt="imgalt">
      <div class="p-2">
        <div class="font-bold text-lg text-gray-
          <a :href="titleurl" class="hover:under
        </div>
        <p class="text-xs leading-tight tracking-
        <div class="flex pt-2 border-t border-gra
          <div class="flex-1 text-center">
            <i class="far fa-clock"></i>
            <p>{{ time }}</p>
          </div>
          <div class="flex-1 text-center">
            <i class="fas fa-utensils"></i>
            <p>{{ servings }}</p>
          </div>
          <div class="flex-1 text-center">
            <i class="fas fa-signal"></i>
            <p>{{ level }}</p>
          </div>
        </div>
      </div>
    </div>`
})

app.mount('#app')
</script>
```

Here, we've put all repeating code in a template and defined props for the recipe's details. This way, we can provide different details for each recipe while the template remains the same. If we decide in the future to change the card's appearance, we'll need to edit its template in only one place.

REBUILDING STYLES

Build Tailwind styles again (if you don't use the `dev:watch` script) for the changes to take effect.

Chapter 3: Building Complex Designs with Tailwind

In the first chapter, we covered the basic concepts of the Tailwind framework and how to use it to build simple designs. We then expanded this knowledge by learning how we can build reusable components and responsive designs.

In this chapter, we'll extend what we've learned with more advanced layout-building and design-enhancement techniques, including creating and using:

- grid layouts
- drop caps
- gradients
- image clipping
- image filters
- transforms and transitions

We'll start with several grid layouts and then we'll build a full article layout.

Building Complex and Flexible Layouts with Tailwind's Grid Utilities

In this section, we'll build several grid layouts similar to magazine or news sites. To do that, we'll use Tailwind's grid utilities.

CSS GRID

If you need to get up to speed with Grid layout in CSS, you can do it quickly by playing this fun CSS Grid Garden game. For more in-depth exploration of CSS Grid and CSS in general, I suggest checking out *CSS Master*.

PROJECT CODE

You can find the finished project for this chapter in the code repo for this book.

Exploring Tailwind's Grid Utilities

Here's a quick overview of the grid utilities we'll use in our layouts.

To define columns and rows, you use the following classes (`n` is the number of columns/rows):

- `grid-cols-{n}` : define columns
- `grid-rows-{n}` : define rows

To add a gap between columns and/or rows, you can use the following classes:

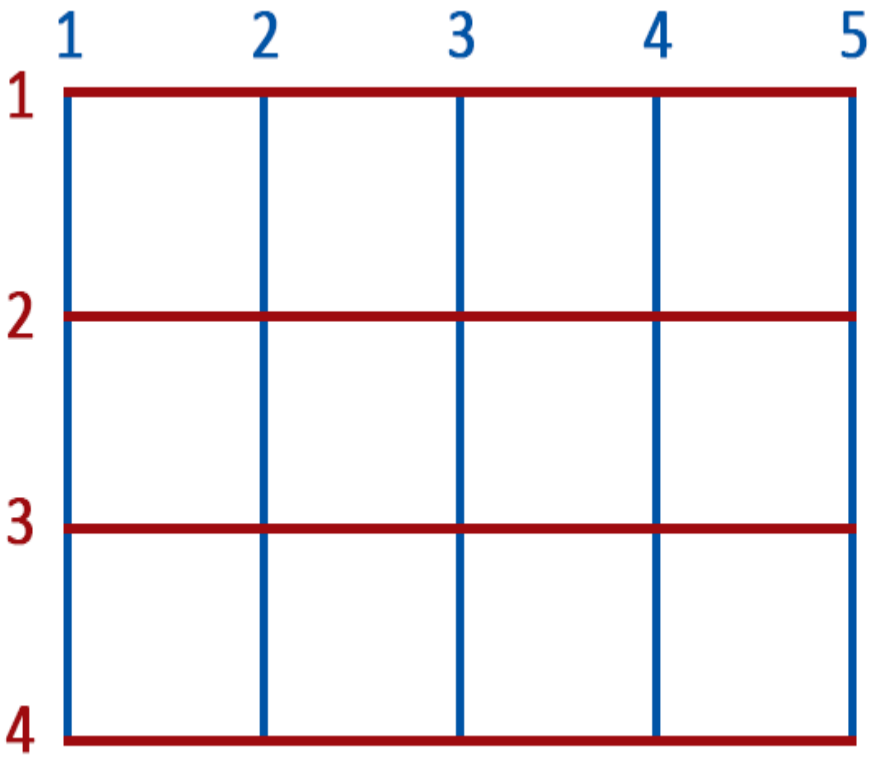
- `gap-{size}` : define gap between columns and rows at the same time
- `gap-x-{size}` : define gap between columns
- `gap-y-{size}` : define gap between rows

To start creating complex layouts, you'll need the following classes:

- `col-span-{n}` : make an element span `n` columns
- `col-start-{n}` : make an element start at grid line `n`
- `col-end-{n}` : make an element end at grid line `n`
- `row-span-{n}` : make an element span `n` rows
- `row-start-{n}` : make an element start at grid line `n`
- `row-end-{n}` : make an element end at grid line `n`

NUMBERING GRID LINES

CSS grid lines start at 1, not 0. So in a four-column, three-row grid, columns would start at column line 1 and end at column line 5, and rows would start at row line 1 and end at row line 4—as illustrated below.



You can also use `order-{order}` to display grid items in a different order from their DOM order.

With the above utilities, we can create an endless number of layouts with different levels of complexity. Let's explore some examples now.

Creating Grid Layouts

We'll borrow most of the following examples from this nice looking Newsportal theme.

You can see the completed grid layout examples in this CodePen demo.

To follow along, let's create a new project, just as we learned how to do in the previous chapter.

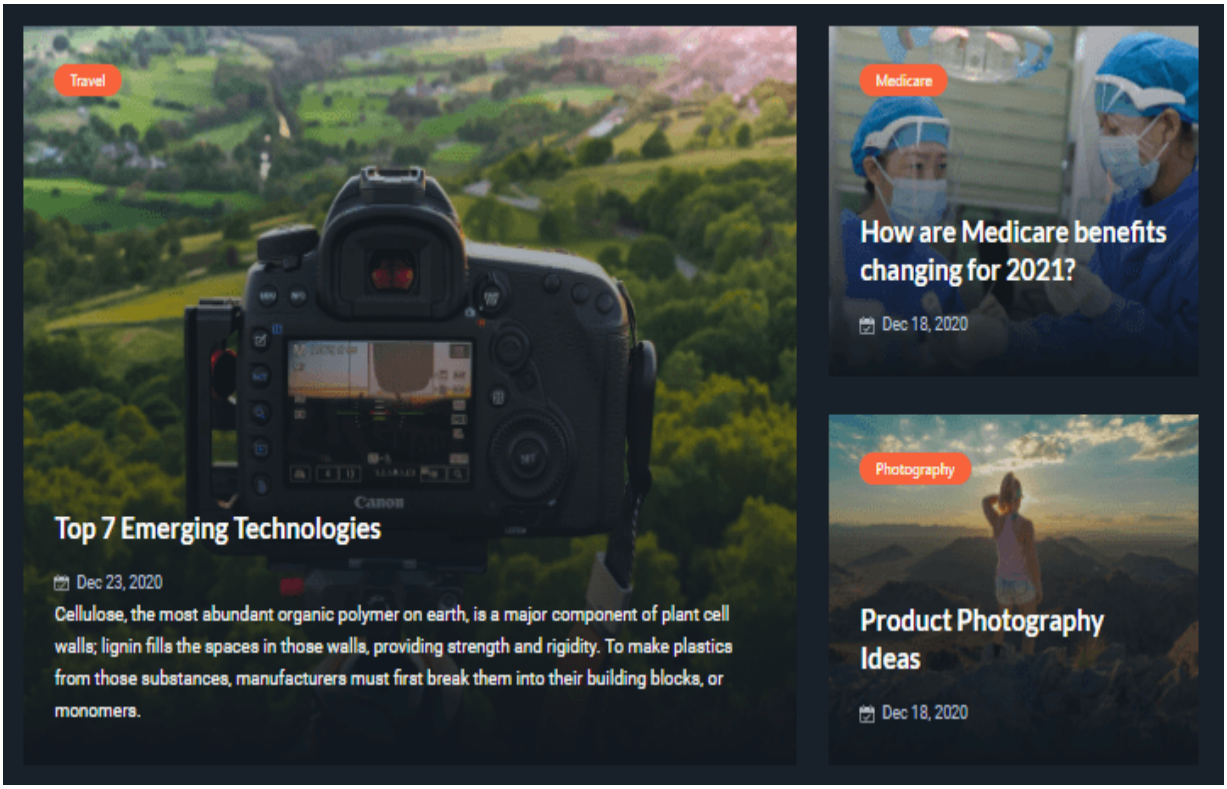
Next, you need to configure Tailwind to use all HTML files, by modifying the `content` section in the `tailwind.config.js` file:

```
// /tailwind.config.js
module.exports = {
  content: ['*.html'],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

Lastly, build the styles by running this command:

```
npm run dev:watch
```

Now, let's start with the first layout. The image below shows what we want to build.



To build the first layout, in the root directory, create a new `grid-s.html` file with the following content:

```
<!-- /grids.html -->
<!doctype html>
<html>
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-w
    <link rel="stylesheet" type="text/css" href="
    <style>
      .box {
        color: white;
        background-color: green;
```

```
        min-height: 150px;
        padding-top: 10%;
        padding-left: 15%;
        font-size: 1.5rem;
        font-weight: 700;
      }
    </style>
  </head>
  <body>
    <div class="mx-auto p-8 w-full lg:w-1/2">

      <h1 class="text-3xl font-bold">Grid Layout

      <h2 class="my-6 text-2xl underline underli

      <div class="grid grid-cols-3 grid-rows-2 ga
        <div class="box row-span-2 col-span-2">1<
        <div class="box">2</div>
        <div class="box">3</div>
      </div>

    <!-- Add the next grid examples here -->

    </div>
  </body>
</html>
```

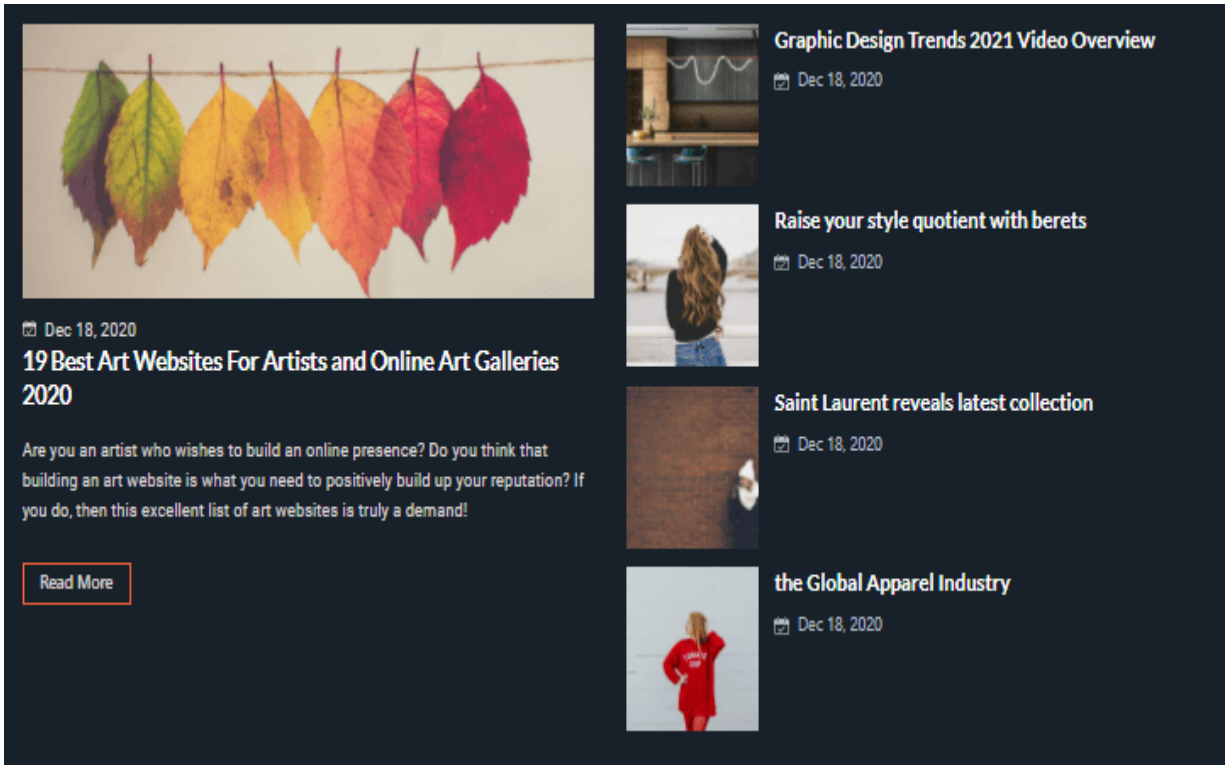
Demo Styles

The `box` class is used here and later just for demonstration purposes, so we can see the visual shape of our grids.

As you can see, we've effectively replicated the design in the earlier screenshot.

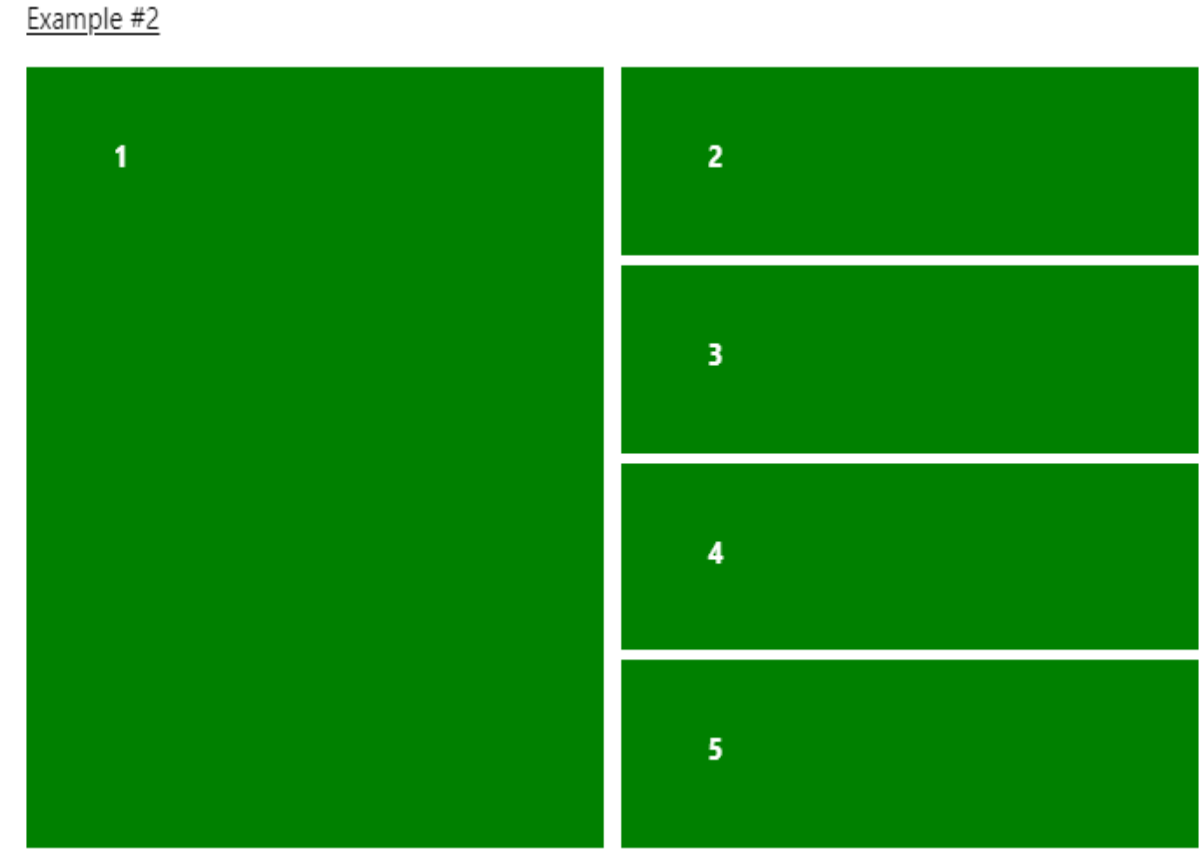
To achieve this, we've firstly created a grid container and specified that our grid will have three columns and two rows. We've also added different gap spaces between columns and rows. Then we've made the first element span two columns and two rows.

Let's move on to the next layout. The final result is shown below.



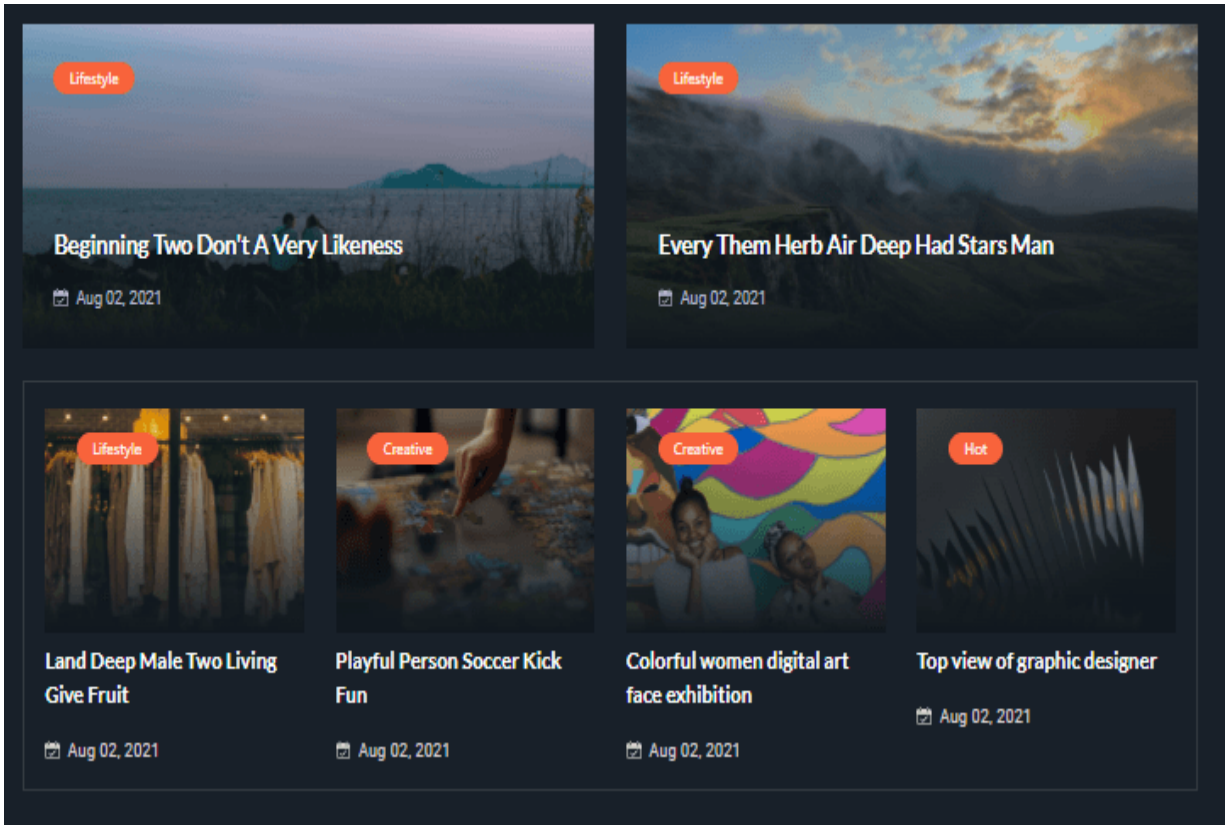
As you can see, this is quite similar to the first example, but under the hood we'll do some different things to achieve it. Here's the code to add after the first example:

```
<!-- /grids.html -->
<h2 class="my-6 text-2xl underline underline-offs

<div class="grid grid-cols-2 grid-rows-4 gap-x-4
  <div class="box row-span-4">1</div>
  <div class="box">2</div>
  <div class="box">3</div>
  <div class="box">4</div>
  <div class="box">5</div>
</div>
```



This time, we've used two columns and four rows, and the only thing we've needed to do is span the first element over four rows. Easy, huh?

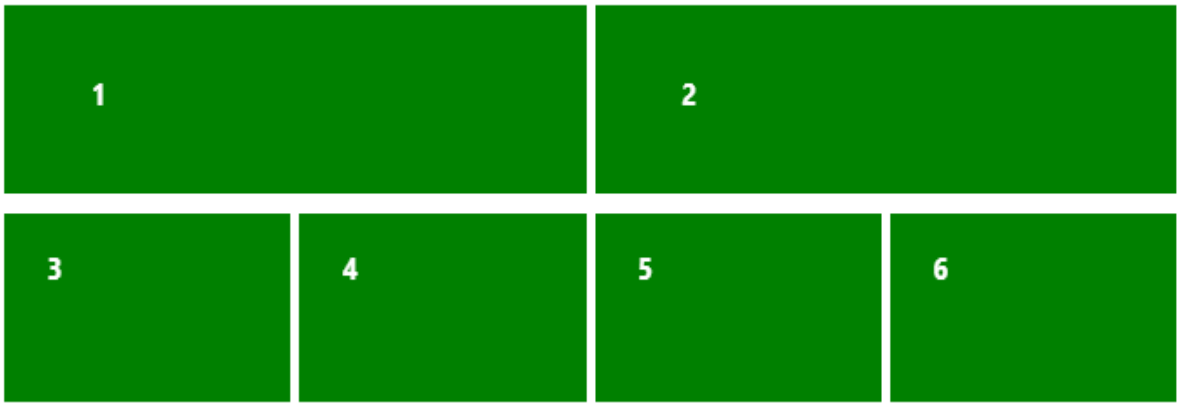
The next layout is quite a bit different. This is what it looks like fully finished.



Here's the code to add after the second example:

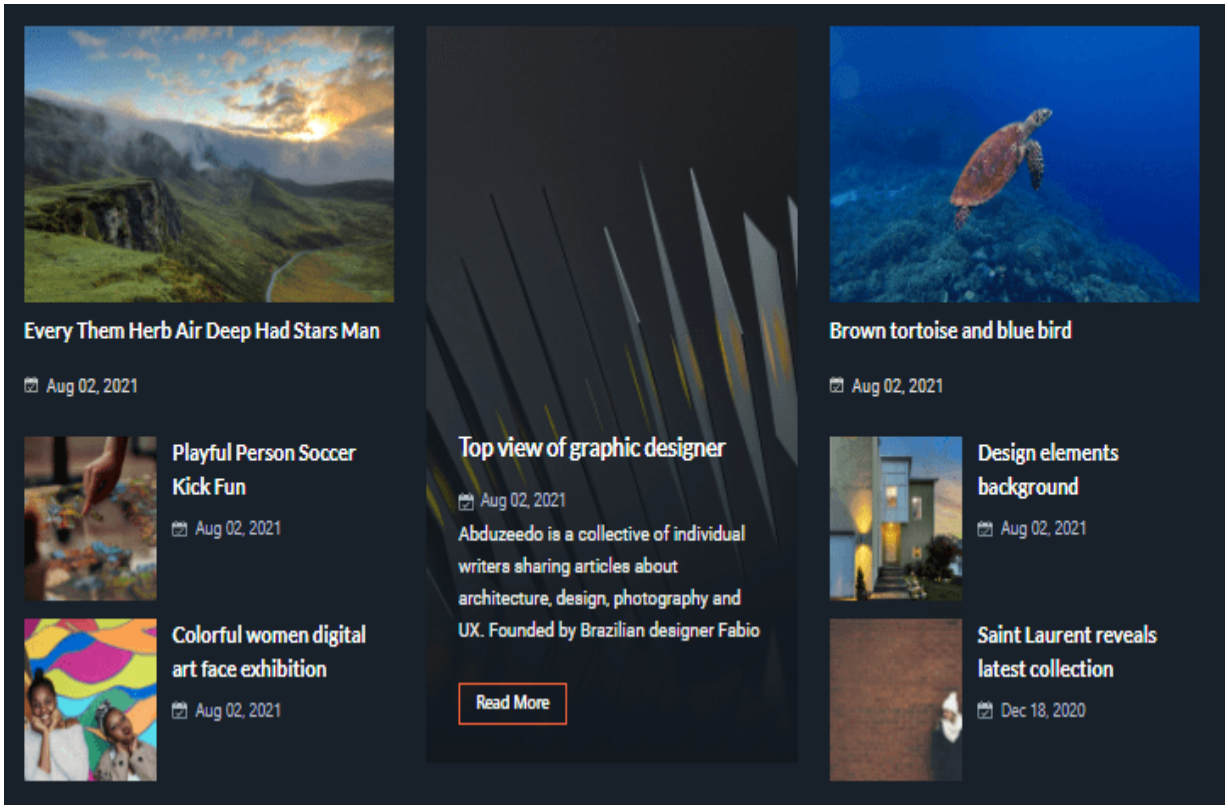
```
<!-- /grids.html -->
<h2 class="my-6 text-2xl underline underline-offs

<div class="grid grid-cols-4 grid-rows-2 gap-x-2
  <div class="box col-span-2">1</div>
  <div class="box col-span-2">2</div>
  <div class="box">3</div>
  <div class="box">4</div>
  <div class="box">5</div>
  <div class="box">6</div>
</div>
```

Example #3



In this example, we've created four columns and two rows for the grid. Then we've just spanned the first two elements over two columns. Once again, it was quite simple to achieve this result.

OK. Let's try something more complex now.

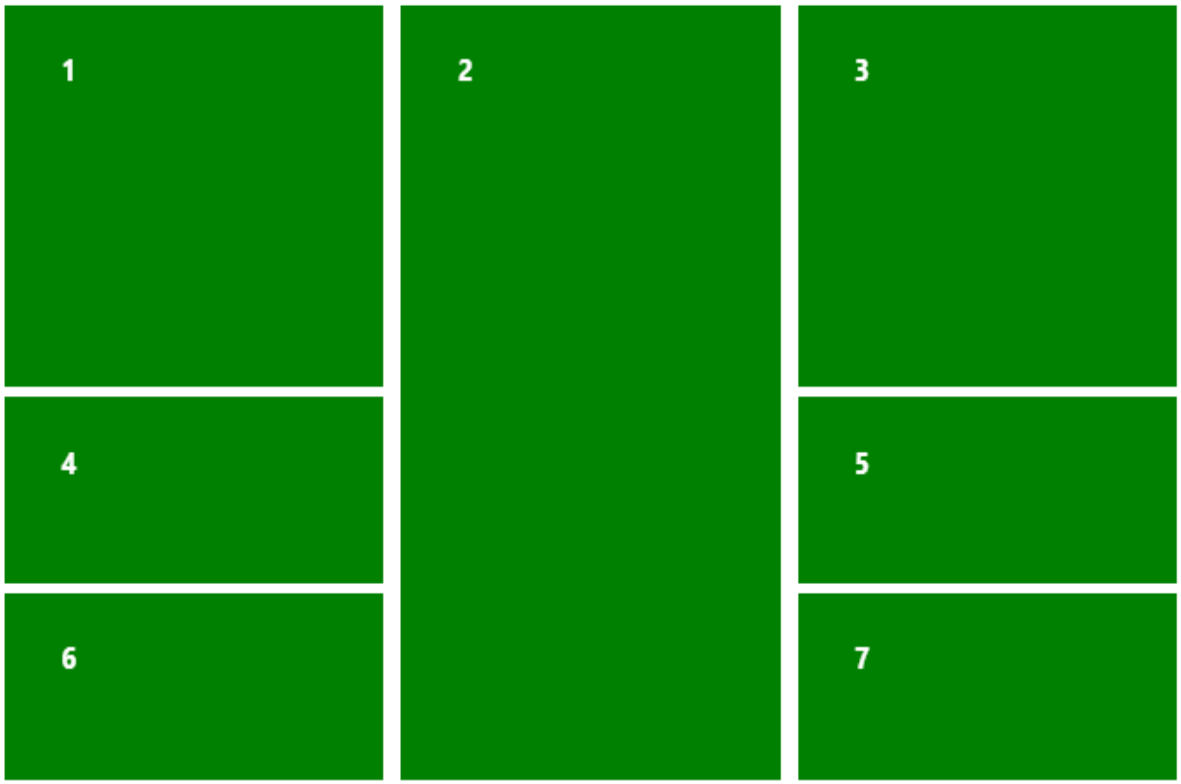


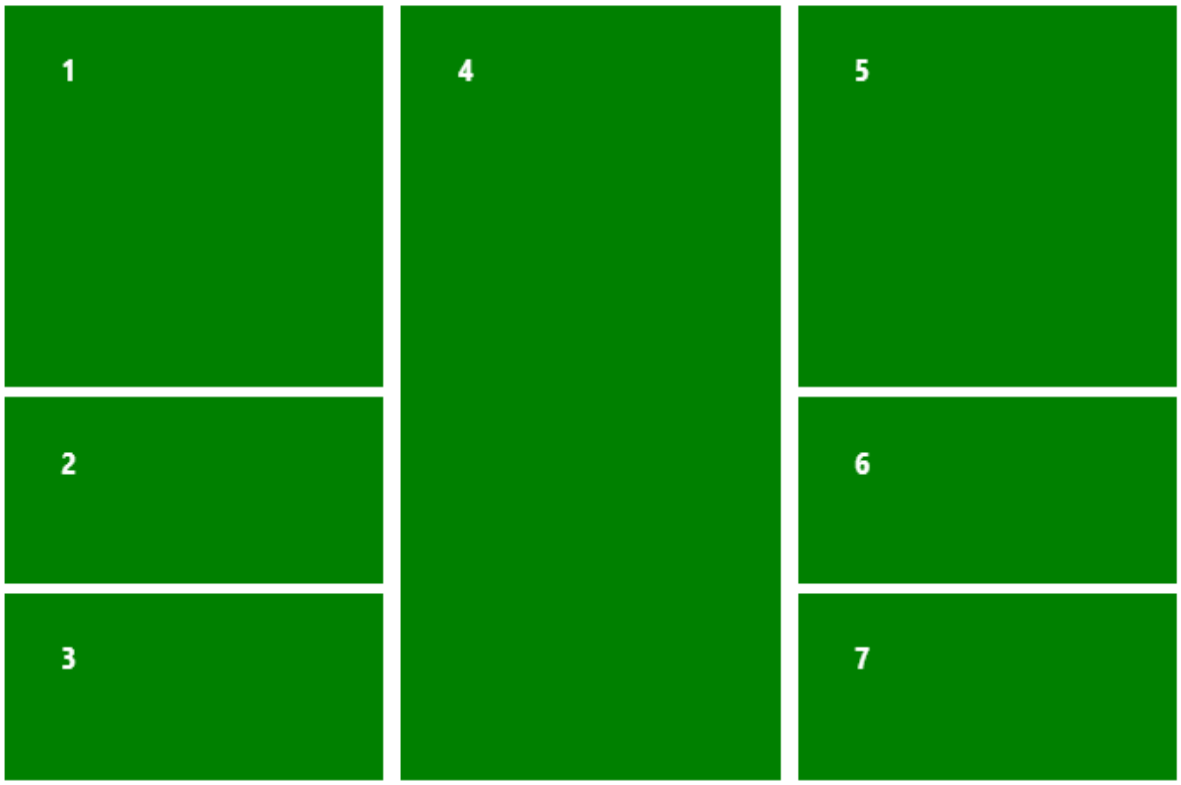
Here's the code to add after the third example:

```
<!-- /grids.html -->
<h2 class="my-6 text-2xl underline underline-offs

<div class="grid grid-cols-3 grid-rows-4 gap-x-4
  <div class="box row-span-2">1</div>
  <div class="box row-span-4">2</div>
  <div class="box row-span-2">3</div>
  <div class="box">4</div>
  <div class="box">5</div>
  <div class="box">6</div>
  <div class="box">7</div>
</div>
```

```
<hr class="my-6">

<div class="grid grid-cols-3 grid-rows-4 gap-x-4
  <div class="box row-span-2">1</div>
  <div class="box row-start-3">2</div>
  <div class="box row-start-4">3</div>
  <div class="box row-span-4">4</div>
  <div class="box row-span-2">5</div>
  <div class="box">6</div>


  <div class="box">7</div>
</div>
```

Example #4





In this example, I've included two variations to show you that there's more than one way to create the same layout.

The first version is the easier to achieve. We've just spanned the first three elements across the desired number of rows. This also creates a sort of row ordering, as you can see from the box numbers. But what if we want the elements to display column by column?

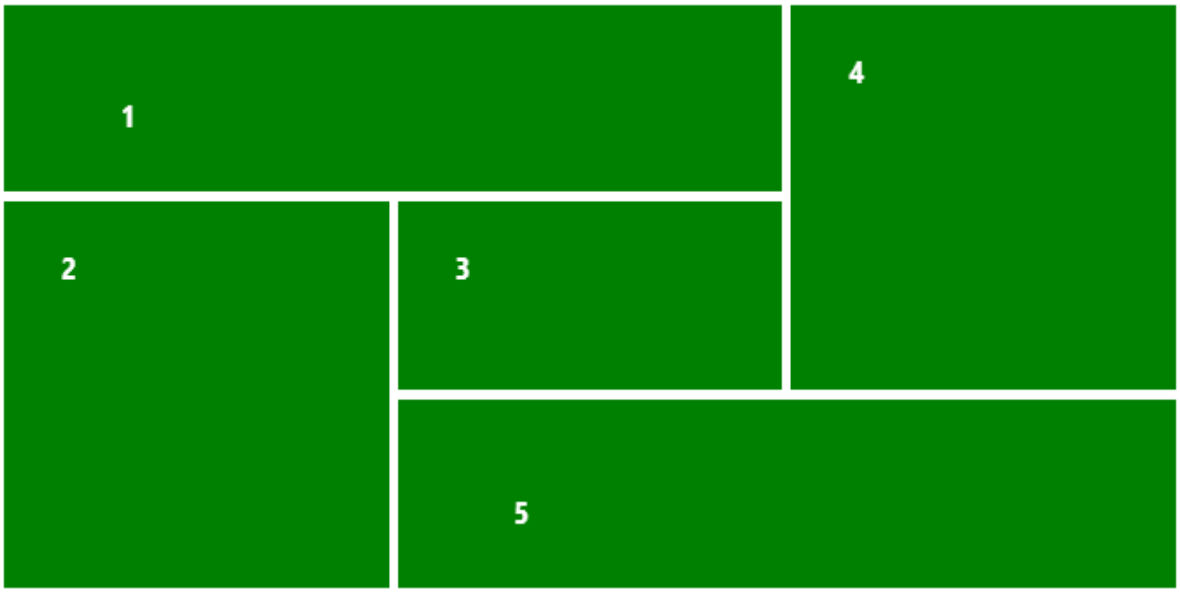
The second version demonstrates how this can be achieved. The first element is easy. We've just spanned it over two rows. For the second, we've used a `row-start-{n}` class to put it in the right place—below the first element. We've used the same technique to put the third element below the second one. The forth and fifth elements are also easy. We've spanned them four and two rows respectively, to put them properly in the grid. The last two elements don't need any classes because they flow naturally into the right places.

As we can see, to create a fairly complex layout is quite easy with these grid utilities. Let's create an even more complex final example. Here's the code to add after the fourth example:

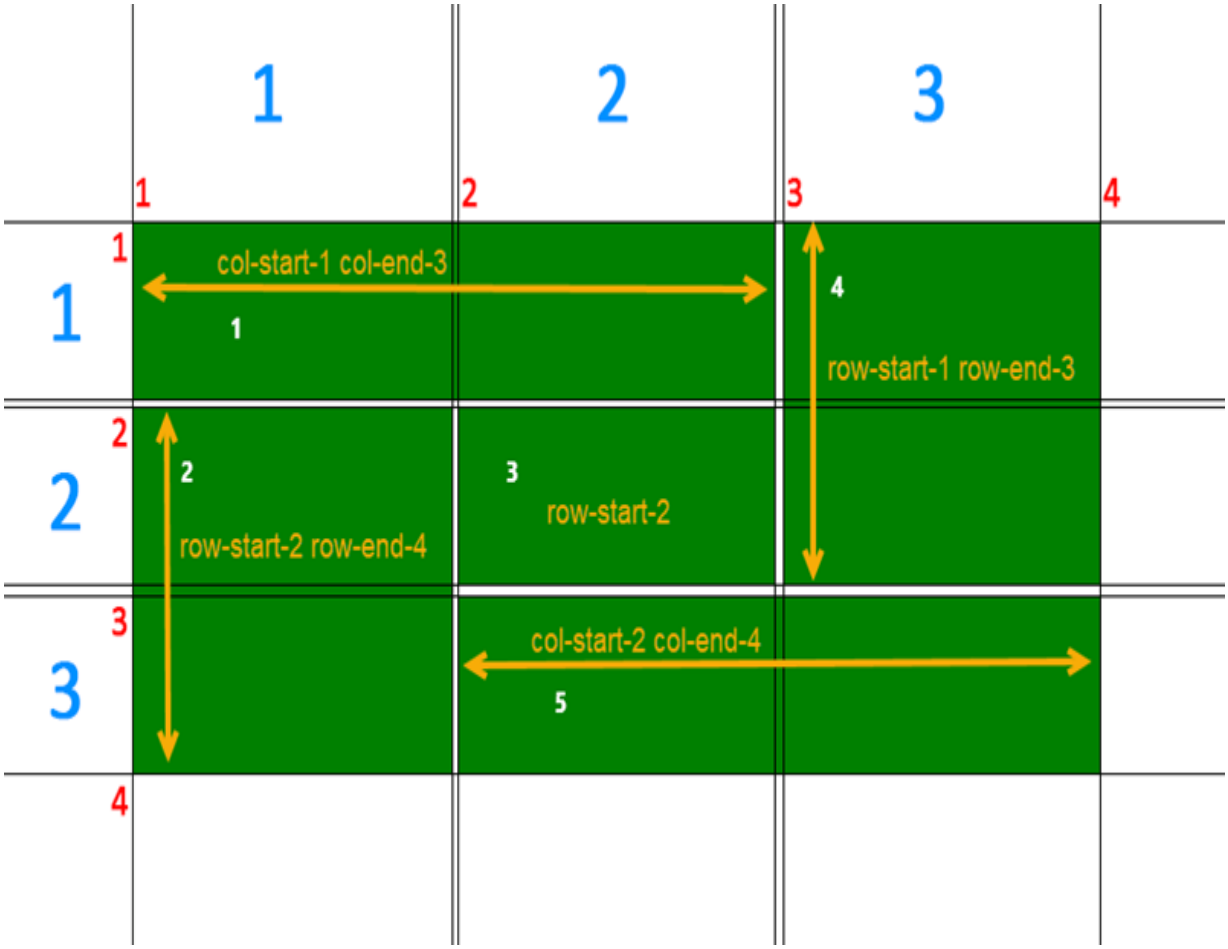
```
<!-- /grids.html -->
<h2 class="my-6 text-2xl underline underline-offs

<div class="grid grid-cols-3 grid-rows-3 gap-2">
```

```
    <div class="box col-start-1 col-end-3">1</div>
    <div class="box row-start-2 row-end-4">2</div>
    <div class="box row-start-2">3</div>
    <div class="box row-start-1 row-end-3">4</div>

    <div class="box col-start-2 col-end-4">5</div>
  </div>
```

Example #5



This creates a very interesting layout. The image below is labeled so we can more easily understand how the code works.



For this example, we've used a grid of three columns and three rows. We've mixed the `start` and `end` classes.

We've told the first element that it should start at the first vertical grid line and end at the third one. We've moved the second element below the first one by making it start at the second horizontal grid line and end at the fourth.

For the third element, we've only needed to define its start line. We've forced the fourth element to start at the first horizontal line and end at

the third. And finally, we've spanned the fifth element over two columns by defining the second vertical grid line as a starting point and the fourth one as the end.

And that's it. It's easy, but it needs careful planning and lots of experimenting.

By the way, we can use a different utility combination to achieve the exact same result. For example, here's a variation that uses mostly `span` utilities instead of `start` and `end`:

```
<div class="grid grid-cols-3 grid-rows-3 gap-2">
  <div class="box col-span-2">1</div>
  <div class="box row-start-2 row-span-2">2</div>
  <div class="box row-start-2">3</div>

  <div class="box row-span-2">4</div>
  <div class="box col-span-2">5</div>
</div>
```

The end result is the same.

As we can see, CSS grids allow us to easily build complex layouts that are almost impossible to build with CSS 2.x without using some sort of dirty hacks and/or complicated workarounds.

Creating a Complete Article Design

In this section, we'll explore how to create a complete article/post design, employing Tailwind utility classes for layout, typography, colors, images, and a bit of interactivity. The final result is shown below.

Lorem ipsum dolor sit amet, consectetur

adipiscing elit. Maecenas varius vitae ipsum et commodo. In scelerisque est magna, ut fringilla purus congue eu. Mauris id metus ac metus porta aliquet. Aliquam quam ipsum, consequat malesuada lectus nec, blandit condimentum enim. Donec varius mattis facilisis. Morbi rhoncus erat vel erat pellentesque suscipit. Nunc dictum euismod libero sed tristique.

> *Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas varius vitae ipsum et commodo. In scelerisque est magna, ut fringilla purus congue eu. Mauris id metus ac metus porta aliquet.*

Aliquam id nulla dignissim felis bibendum aliquam. Cras vulputate blandit semper. Nam quam dolor, tincidunt non odio ac, condimentum molestie justo. In ornare maximus tortor, aliquam consequat arcu sagittis id. Vivamus condimentum varius ante, pulvinar laoreet tortor dignissim ac. Curabitur egestas in arcu sit amet feugiat. Aenean interdum, purus eget sodales tincidunt, magna sem lobortis nunc, at porttitor arcu velit sed augue. Praesent porttitor nisl enim, eget sollicitudin ipsum porta a. Integer tincidunt, lorem sit amet gravida hendrerit, arcu felis convallis metus, vitae vehicula diam dolor vulputate sapien. Nulla finibus lectus nec porta faucibus. Praesent in massa sollicitudin, dignissim quam at, volutpat mauris. Morbi in turpis sapien. Morbi ante est, gravida vitae ipsum quis, pretium scelerisque massa.

Vestibulum congue felis at posuere commodo. Praesent sapien magna, aliquet ut efficitur et, luctus at neque. Donec vitae nunc convallis, maximus ex sit amet, consequat tellus. Aenean eleifend cursus urna, sed fermentum felis cursus eget. Sed accumsan hendrerit turpis at ullamcorper. Integer quam sapien, rutrum ac pharetra eget, maximus id lacus. Vivamus sollicitudin molestie hendrerit. Proin sem quam, tempus in felis et, elementum dignissim felis. Integer odio tellus, semper at commodo at, euismod id magna. Aenean nisl metus, maximus a eros sit amet, facilisis malesuada lorem. Mauris est tortor, accumsan ac aliquet et, sagittis vitae velit. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos.

Aenean scelerisque urna id dictum tempor. Pellentesque ipsum orci, convallis eget purus nec, placerat laoreet nulla. Nullam vitae lectus porta, lacinia neque at, rutrum felis. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Morbi luctus, magna eget gravida lobortis, elit elit rutrum urna, at volutpat ex orci accumsan urna. Sed et ligula magna. Ut dignissim semper ligula, at lacinia mi dignissim non. In eleifend ultricies viverra. Nunc euismod ac lacus ac molestie. Nam sem lectus, malesuada a ipsum vitae, viverra condimentum elit. Ut at vulputate tortor, nec suscipit leo. In non dolor nec purus semper tempus:

- First item
- Second item
- Third item

Pellentesque tincidunt non orci id congue. Donec blandit pulvinar leo et tincidunt. Sed venenatis venenatis justo, ut congue neque lobortis sit amet. Nam tempus vehicula nisi, vitae commodo magna condimentum id. In quis vehicula massa. Fusce id congue lorem. Duis imperdiet placerat metus, vitae hendrerit lorem sollicitudin sit amet. Integer varius justo non velit semper elementum. Donec scelerisque magna nibh, at efficitur elit hendrerit id. Aenean sagittis lectus odio, eu varius lectus vestibulum eget. Sed gravida mattis auctor.

David Smith

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas varius vitae ipsum et commodo.

Creating the Base Layout

OK. Let's do some coding. In the root directory, create a new `article.html` file with the following content:

```
<!-- /article.html -->
<!doctype html>
<html>
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-w
    <link rel="stylesheet" type="text/css" href="
  </head>
  <body>
    <div class="m-6 md:mx-auto p-8 space-y-6 md:w
      <p>Lorem ipsum dolor sit amet, consectetur
      <quote class="px-4 py-2 block border-l-4">I
      <p>Aliquam id nulla dignissim felis bibendu
      <img class="mr-3 w-1/2 float-left rounded"
      <p>Vestibulum congue felis at posuere commo
      <p>Aenean scelerisque urna id dictum tempor
      <ul class="ml-6 pl-6 space-y-3">
        <li>First item</li>
        <li>Second item</li>
        <li>Third item</li>
      </ul>
      <p>Pellentesque tincidunt non orci id cong
```

```
        <p>Pellentesque tincidunt non orci id congu
        <hr>
        <div class="flex pt-6">
          <img class="mr-6 mb-6 w-24 h-24 rounded b
          <div>
            <p class="-mt-3">David Smith</p>
            <p class="mt-2">Lorem ipsum dolor sit a
          </div>
        </div>
      </div>
    </body>
  </html>
```

Here's a live version of that code on CodePen.

ABBREVIATED TEXT

For brevity's sake, from now on I won't use the full paragraph text. But you need to use the full text to display the example correctly.

We'll start by creating an article container:

```
  <div class="m-6 md:mx-auto p-8 space-y-6 md:w-1/2

  </div>
```

We've added some visual space (`space-y-6`) between all direct children elements. Then we've added two width modifiers (`md:w-1/2 xl:w-1/3`) that will produce the following effect: the article will start at full width at small screens (this is by default) up to the medium screens, where the width will change to one and a half. This setting will prevail up to the extra large screens and above, where the width will change to one third. Also starting from medium screens and above, the article will be centered, thanks to the `md:mx-auto` property.

Next, we'll add the article content:

```
<p>Lorem ipsum dolor..</p>
<quote class="px-4 py-2 block border-l-4">Lorem i
<p>Aliquam id nulla..</p>
<img class="mr-3 w-1/2 float-left rounded" src="h
<p>Vestibulum congue felis..</p>
<p>Aenean scelerisque urna.. In non dolor nec pur
<ul class="ml-6 pl-6 space-y-3">
  <li>First item</li>
  <li>Second item</li>
  <li>Third item</li>
</ul>
<p>Pellentesque tincidunt non..</p>
```

Here, we've added the following elements:

- a paragraph
- a quote with basic styling and border
- another paragraph
- an rounded and floated left image with half width
- another two paragraphs
- a list
- and the last paragraph

Finally, we'll add the author section:

```
<hr>
<div class="flex pt-6">
  <img class="mr-6 mb-6 w-24 h-24 rounded border-
  <div>
    <p class="-mt-3">David Smith</p>
    <p class="mt-2">Lorem ipsum dolor sit amet, c
  </div>
</div>
```

Here, we've added an `hr` element to divide the author section from the article content. We've then wrapped the author info in a flex container. We've styled the image with a polaroid-like effect by using bor-

der classes. To achieve this, we've used an arbitrary value of `16px` for the bottom border. To style the link, we've used the `underline` `underline-offset-1` classes, which add a line with a small offset.

We now have a base layout that we can build upon. Let's continue developing the article design by adding some typographical features.

Typography

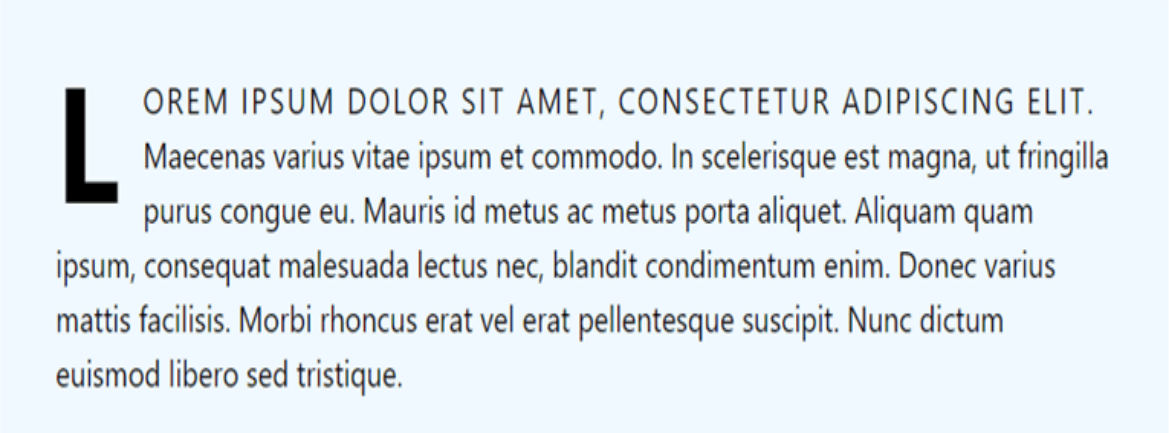
In this section, we'll explore typographical features such as drop caps. Here's what the code should look like after the additions:

```
<!-- /article.html -->
<div class="m-6 md:mx-auto p-8 space-y-6 md:w-1/2
  <p class="first-letter:-mt-2 first-letter:mr-3
  <quote class="px-4 py-2 block border-l-4 font-l
  <p class="indent-6">Aliquam id nulla..</p>
  <img class="mr-3 w-1/2 float-left rounded" src=
  <p class="indent-6">Vestibulum congue felis..</
  <p class="indent-6">Aenean scelerisque urna.. I
  <ul class="ml-6 pl-6 space-y-3 list-disc">
    <li>First item</li>
    <li>Second item</li>
    <li>Third item</li>
  </ul>
```

```
    <p class="indent-6">Pellentesque tincidunt non
    <hr>
    <div class="flex pt-6">
      <img class="mr-6 mb-6 w-24 h-24 rounded borde
      <div>
        <p class="-mt-3 font-semibold">David Smith<
        <p class="mt-2 text-sm">Lorem ipsum dolor s
      </div>
    </div>
  </div>
```

This CodePen demo shows our typography settings in action.

To create a drop cap and style the first line of the paragraph, we've used the `first-letter` and `first-line` pseudo elements (`first-letter:-mt-2 first-letter:mr-3 first-letter:float-left first-letter:text-7xl first-letter:font-bold first-line:uppercase first-line:tracking-widest`).



We've also added some indent (`indent-6`) to all paragraphs except the first one for easier reading.

We've made the text for the quote thin and italic (`font-light italic`).

We've added a disc symbol (`list-disc`) for the list item bullets.

We've made the font for the author's name semibold (`font-semibold`) and the text for author's info small (`text-sm`). Finally, we've decorated the link by changing its thickness and style (`decoration-1 decoration-wavy`).

The article is starting to look a lot nicer, but it's still is missing a bit of liveliness. Let's fix that by adding some colors and gradients.

Colors and Gradients

We know already the base use of colors, but now we'll explore how to use gradients too. Here's the code with added colors:

```
<!-- /article.html -->
<div class="m-6 md:mx-auto p-8 space-y-6 md:w-1/2
  <p class="first-letter:-mt-2 first-letter:mr-3
  <quote class="px-4 py-2 block border-l-4 borde
  <p class="indent-6">Aliquam id nulla..</p>
  <img class="mr-3 w-1/2 float-left rounded" src=
  <p class="indent-6">Vestibulum congue felis..</
  <p class="indent-6">Aenean scelerisque urna.. I
  <ul class="ml-6 pl-6 space-y-3 list-disc marke
    <li>First item</li>
    <li>Second item</li>
    <li>Third item</li>
  </ul>
  <p class="indent-6">Pellentesque tincidunt non
  <hr class="text-cyan-600">
  <div class="flex pt-6">
    <img class="mr-6 mb-6 w-24 h-24 rounded borde
    <div>
      <p class="-mt-3 font-semibold text-stone-90
      <p class="mt-2 text-sm">Lorem ipsum dolor s
    </div>

  </div>
</div>
```

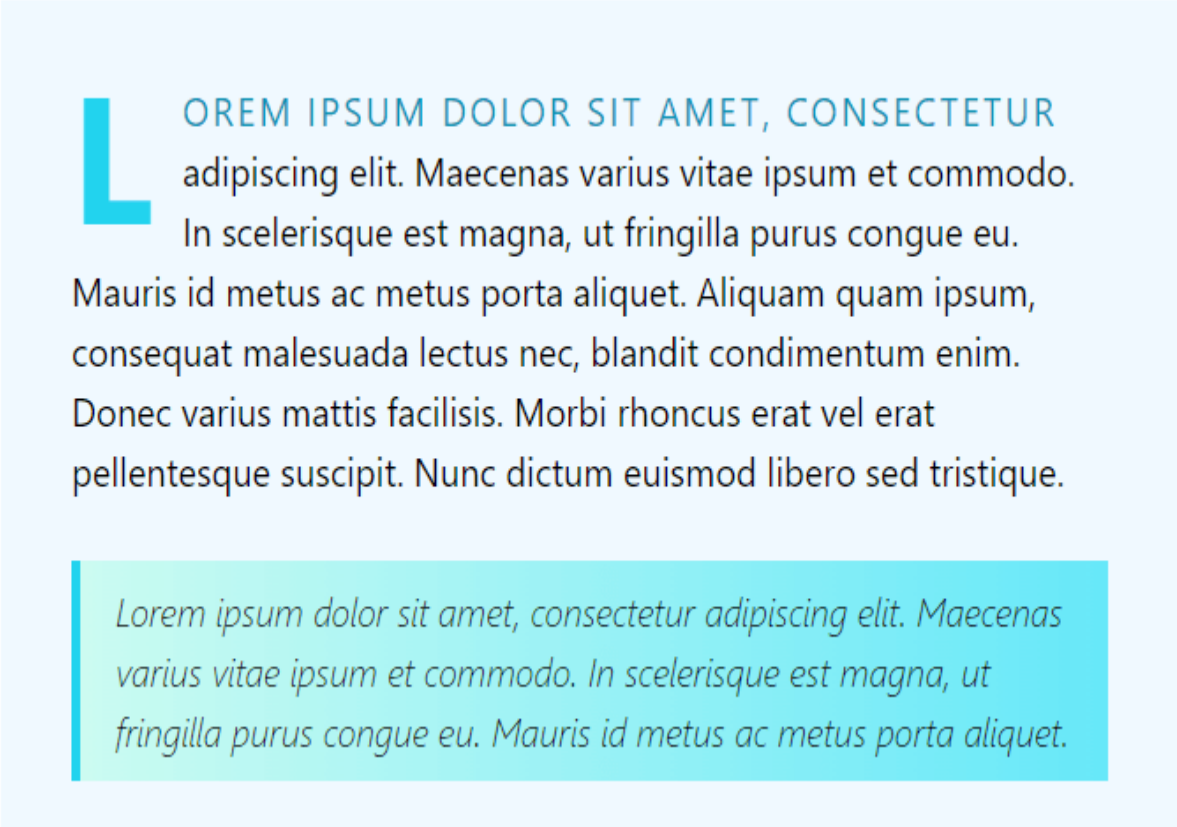
You can see these color changes on this CodePen demo.

Firstly, we've added colors for the drop cap (`first-letter:text-cyan-400`) and the first line of the paragraph (`first-line:text-cyan-600`). We've also added color for the quote element's border (`border-cyan-400`) and a gradient for the body (`bg-gradient-to-r from-teal-100 to-cyan-300`).

The pattern for making gradients is `from-{color} via-{color} to-{color}` . Let's break it down:

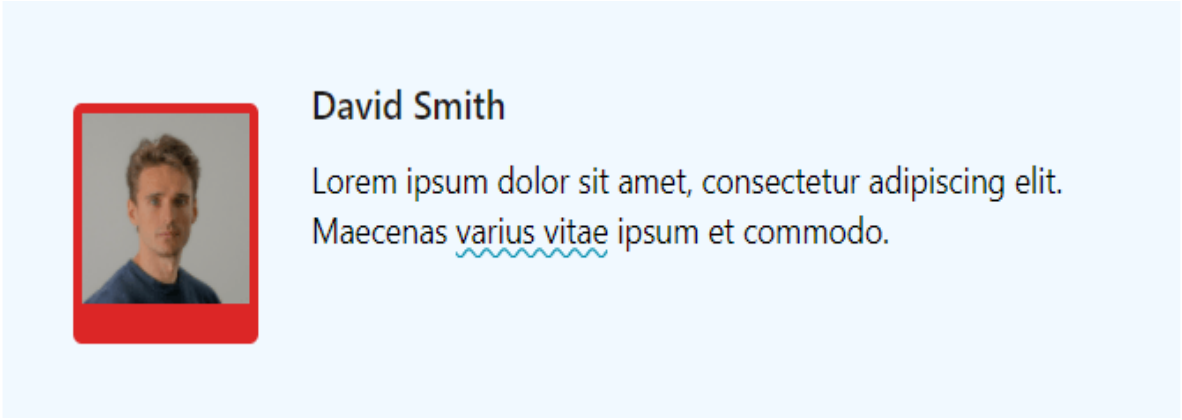
- `from-{color}` : set the starting color of a gradient
- `via-{color}` : add a middle color (or colors) to a gradient
- `to-{color}` : set the ending color of a gradient

The `bg-gradient-to-r` class defines the direction of the gradient —from left to right.



Next, we've added color to the list items (`text-gray-600`) and their bullets by using the `marker` pseudo element (`marker:text-cyan-400`). We've also colored the `hr` element's line (`text-cyan-600`) to fit the article theme.

We've made the author image border red (`border-red-600`) and author name next to it dark brown (`text-stone-900`). And the last color change is to the link. We've colored the underline and changed the text color on hover (`decoration-cyan-600 hover:text-cyan-600`).



And that's it. Now our article design looks much more vivid and alive. To extend this further, let's now enhance the appearance of the images with some effects.

Adding Image Effects

In this section, we'll explore how you can make your images more visually appealing. We'll add sepia and drop-shadow effects by using the corresponding Tailwind filters. Besides that, we'll use arbitrary values to clip the image and make text flow around it. Here's the code with image effects added:

```
<!-- /article.html -->
<div class="m-6 md:mx-auto p-8 space-y-6 md:w-1/2
...
  <img class="mr-3 w-1/2 float-left rounded hover
  <p class="indent-6">Vestibulum congue felis..</
...
  <div class="flex pt-6">
```

```
      <img class="mr-6 mb-6 w-24 h-24 rounded borde
      <div>
        <p class="-mt-3 font-semibold text-stone-9(
        <p class="mt-2 text-sm">Lorem ipsum dolor s
      </div>
    </div>
  </div>
```

You can see the code live in this CodePen demo.

For the article image, we've firstly added two filters that will take effect on hover (`hover:sepia hover:drop-shadow-lg`). Then we've used the power of arbitrary values to clip the image (`[clip-path:circle(80%_at_30%_20%)]`) and force the text to flow around it (`[shape-outside:circle(80%_at_30%_20%)]`).

Vestibulum congue felis at posuere commodo. Praesent sapien magna, aliquet ut efficitur et, luctus at neque. Donec vitae nunc convallis, maximus ex sit amet, consequat tellus. Aenean eleifend cursus urna, sed fermentum felis cursus eget. Sed accumsan hendrerit turpis at ullamcorper. Integer quam sapien, rutrum ac pharetra eget, maximus id lacus. Vivamus sollicitudin molestie hendrerit. Proin sem quam, tempus in felis et, elementum dignissim felis. Integer odio tellus, semper at commodo at, euismod id magna. Aenean nisl metus, maximus a eros sit amet, facilisis malesuada lorem. Mauris est tortor, accumsan ac aliquet et, sagittis vitae velit. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos.

For the author image, we've used a regular shadow (`shadow-md`) with a dark red color (`shadow-red-900`) for a more natural look.

Now the article looks even more attractive, but let's take it to the limit by adding some transforms and transitions.

Adding Effects

In this final section, we'll see how to add some bells and whistles to our article design. Here's the code with the effects added:

```
<!-- /article.html -->
<div class="m-6 md:mx-auto p-8 space-y-6 md:w-1/2
...
  <div class="flex pt-6">
    <img class="mr-6 mb-6 w-24 h-24 rounded borde
    <div>
      <p class="-mt-3 font-semibold text-stone-90
      <p class="mt-2 text-sm">Lorem ipsum dolor s
    </div>
  </div>
</div>
```

You can see these effects live in this CodePen demo.

Here are the classes we've added to the author image (`origin-bottom-left -rotate-6 hover:rotate-0 hover:scale-110 transition duration-500`). We've set the transform origin to be in the bottom left of the image. We've then rotated the image slightly. When the image is hovered over, the image will be rotated to its normal position and scaled up slightly. Finally, we've specified that the transform property should transition when it's changed (`transition-transform`), and we've also set the transition duration (`duration-500`).

Great! Now our article design is complete and has a compelling look and feel.

Conclusion

So far, we've learned how to build complex layout designs with Tailwind's grid utilities, and how we can build a complete article design by combining many of Tailwind's utilities for layout, typography, colors, imagery, and interactivity.

To further explore and build on your Tailwind skills, I suggest you to try to make different variants of the designs we've explored here and experiment with your own designs too.

In the next chapter, we'll explore various ways to customize Tailwind by overriding or extending the base styles and default theme, and ways to create reusable configuration presets.

Chapter 4: Customizing Tailwind and Optimizing Your Workflow

So far we've explored various ways to use the existing Tailwind utilities. In this fourth chapter, we'll dive in even deeper and explore how we can customize Tailwind either by adding new utilities or by tweaking existing ones.

Customizing Tailwind

Tailwind is already like a CSS Swiss Army knife, but nevertheless there will be times when we'll want to add extra features to it. In this section, we'll explore the most common ways Tailwind can be customized to suits our needs.

PROJECT CODE

You can find the finished project for this chapter in the code repo for this book.

Firstly, we need to create a new Tailwind project (which we covered in the second chapter).

Next, create an `index.html` file in the root directory and add the following content to it:

```
<!-- /index.html -->
<!doctype html>
<html>
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-wid
  <link href="tailwind.css" rel="stylesheet">
  <link href="https://cdnjs.cloudflare.com/ajax/l
  <link href="https://fonts.googleapis.com/css2?f
</head>
<body>

</body>
</html>
```

We have a bunch of links here:

- The first link includes the compiled Tailwind styles.
- The second link includes the Font Awesome icons library. We'll use its icons in the examples later on.
- The third link includes the Carter One font from the Google Fonts collection. We'll incorporate this font into the examples later on.

The image below shows what this font looks like.

To demonstrate Tailwind's customization features, we'll reuse the responsive header that we created in the second chapter. We'll copy the header section from CodePen and add it to the `index.html` file:

```
<!-- /index.html -->
<!doctype html>
<html>
  ...
```

```
<body>
  <header class="flex items-center justify-betwee
    <div class="flex-shrink-0 ml-6">
      <a href="#">
        <i class="fas fa-drafting-compass fa-2x t
        <span class="ml-1 text-3xl text-blue-200
      </a>
    </div>

    <button id="nav-toggle" class="md:hidden p-2
      <i class="fas fa-bars fa-2x"></i>
    </button>

    <div class="pl-6 w-full md:w-auto hidden md:b
      <ul class="md:flex">
        <li class="mr-6 p-1 md:border-b-2 border-
          <a class="text-blue-200 cursor-default'
        </li>
        <li class="mr-6 p-1">
          <a class="text-white hover:text-blue-30
        </li>
        <li class="mr-6 p-1">
          <a class="text-white hover:text-blue-30
        </li>
        <li class="mr-6 p-1">
          <a class="text-white hover:text-blue-30
        </li>
        <li class="mr-6 p-1">
```

```
            <a class="text-white hover:text-blue-3
          </li>
          <li class="mr-6 p-1">
            <a class="text-white hover:text-blue-3

          </li>
        </ul>
      </div>
    </header>

  <script>
    document.getElementById('nav-toggle').onclick =
      document.getElementById("nav-content").classL
    }
  </script>
  </body>
  </html>
```

We need to remember to add the script part too, as otherwise the menu won't work.

Now we're ready to dive into the actual customization.

Customizing the Default Tailwind Theme

As we saw in the second chapter, to create a Tailwind configuration file we run this command:

```
npx tailwindcss init
```

Doing so creates the following file:

```
// tailwind.config.js
module.exports = {
  content: [],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

We've already explored the `content` key. Now we'll focus on the `theme` key. The `plugins` key will be explored in the next chapter.

The default Tailwind theme can be either overridden or extended, or both. This gives us a great amount of flexibility.

The `theme` key allows us to customize four base things: `screens`, `colors`, `spacing`, and core plugins such as `borderRadius`, `fontFamily`, and so on. We'll explore each one next, starting with the screens.

Customizing Tailwind Theme's Responsive Breakpoint Modifiers

The default responsive utility variants can be overridden by adding our own under the `screens` key like this:

```
module.exports = {
  theme: {
    screens: {
      'sm': '576px',
      'md': '960px',
      'lg': '1440px',
    }
  }
}
```

In this case, the default variants are discarded and won't be available along with the newly added ones.

If we want to keep the existing breakpoints and to extend them with one or more variants, we have two options.

First, if we want to add larger variants, we just add them under the `extend` key:

```
module.exports = {
  theme: {
    extend: {
      screens: {
        '3xl': '1600px',
      }
    }
  }
}
```

Here, the default variants are kept and the new one is added to them. This approach can be used also to override a single breakpoint. In such a case, we use one of the default names and replace its value with a new one. For example:

```
module.exports = {
  theme: {
    extend: {
      screens: {
        'md': '960px',
      }
    }
  }
}
```

Here, the `md` variant's value is replaced, while the rest of the variants keep their default values.

Second, if we want to add smaller breakpoints, things get a bit more complicated. In such a case, we first need to add our smaller breakpoints, and then we must provide the default utilities after them like so:

```
const defaultTheme = require('tailwindcss/default

module.exports = {
  theme: {
    screens: {
      'xs': '475px',
      ...defaultTheme.screens,
    }
  }
}
```

Here, we first import the default theme and use its `screens` key to include the default responsive utilities after the `xs` variant. You may notice that we don't use the `extend` key here. So in fact, we "extend" the breakpoints by redefining them. This is because, if we use the `extend` key, smaller breakpoints will be added to the end of the

list and the order from smallest to largest will be incorrect. In such a case, the breakpoints won't work as expected.

Customizing Tailwind's Theme Colors

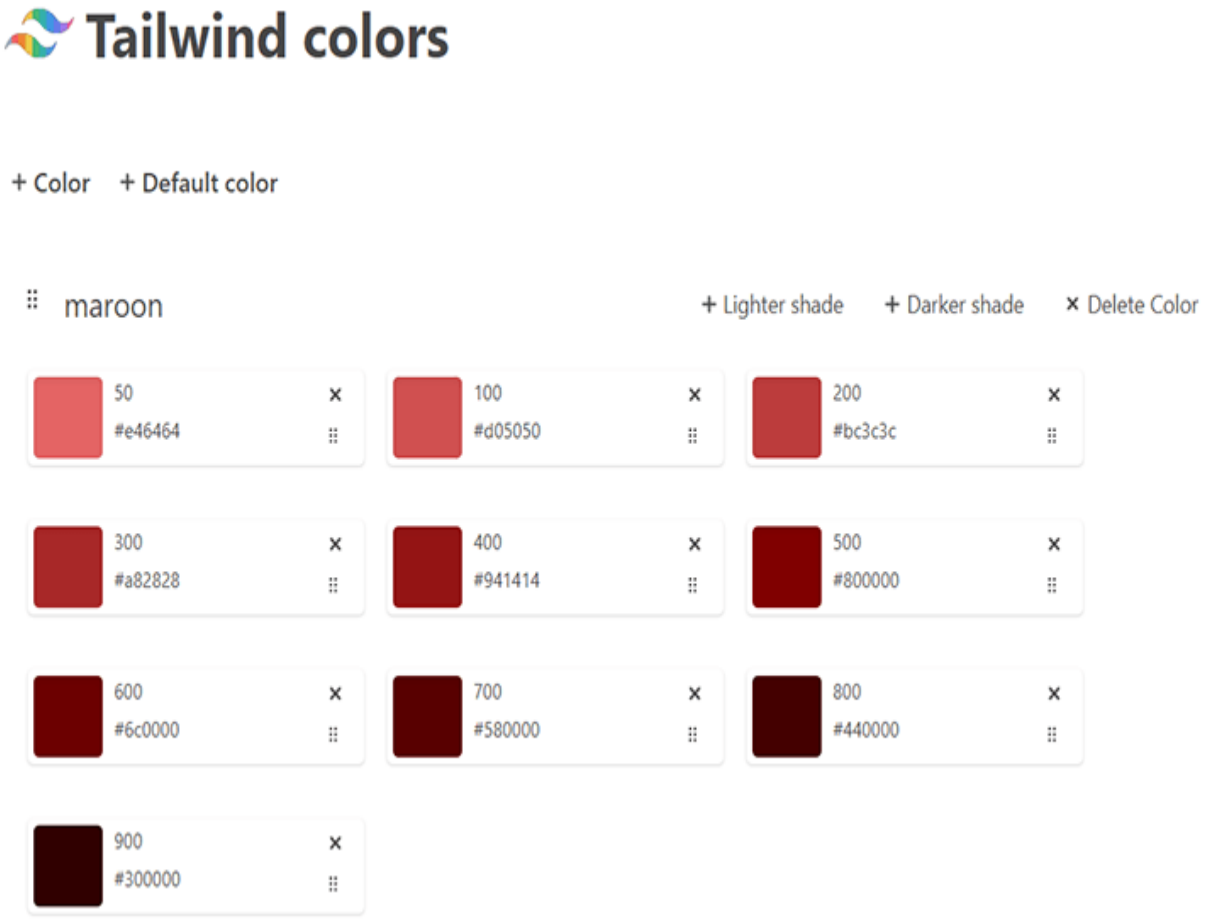
Tailwind offers a precisely selected color palette that will be enough in many scenarios. However, in some cases, we might want to add some specific colors or shades to it—such as our brand colors. In this case, we can extend the default colors like so:

```
module.exports = {
  theme: {
    extend: {
      colors: {
        maroon: {
          50: '#e46464',
          100: '#d05050',
          200: '#bc3c3c',
          300: '#a82828',
          400: '#941414',
          500: '#800000',
          600: '#6c0000',
          700: '#580000',
          800: '#440000',
          900: '#300000'
        },
        indigo: {
```

```
            950: '#1d1a6d'
          }
        }
      }
    }
  }
```

Here, we've added our new `maroon` color shades to the default colors, and also extended the default `indigo` color with one more darker shade.

The image below shows what the maroon color palette looks like.



To generate the palette, I've used the Tailwind colors online tool, which automatically generates the required code.

If we want to completely replace the default Tailwind color palette with our own custom colors, we can do it this way:

```
module.exports = {
  theme: {
    colors: {
      transparent: 'transparent',
```

```
        current: 'currentColor',
        'white': '#ffffff',
        'black': '#000000',
        'tahiti': {
          100: '#cffafe',
          200: '#a5f3fc',
          300: '#67e8f9',
          400: '#22d3ee',
          500: '#06b6d4',
          600: '#0891b2',
          700: '#0e7490',
          800: '#155e75',
          900: '#164e63',
        }
      }
    }
  }
```

Here, we've added simple white and black colors and `tahiti` color shades. We've also included values like `transparent` and `currentColor` in case we want to use them in our project.

If we want to use some of Tailwind's default colors, we can do so by importing them and using the ones we need like so:

```
const colors = require('tailwindcss/colors')
```

```
module.exports = {
  theme: {
    colors: {
      transparent: 'transparent',
      current: 'currentColor',
      black: colors.black,
      white: colors.white,
      gray: colors.gray,
      emerald: colors.emerald,
      indigo: colors.indigo,
      yellow: colors.yellow,
    },
  },
}
```

Now you can use your colors as usual—for example, `text-yellow-500`, `bg-indigo-300`, and so on.

NAMING COLORS

We can use different names for our colors if we wish. For example, `green: colors.emerald`. In this scenario, we would use it like this: `text-green-400`, `bg-green-700`, and so on.

Customizing Tailwind's Spacing Utilities

Tailwind has a rich set of spacing utilities—which are detailed in the documentation on Tailwind's default spacing scale.

However, sometimes we might need a bit more precision. In such a situation, we can add the needed utilities again in two ways.

We can just discard the Tailwind utilities and replace them with our own:

```
module.exports = {
  theme: {
    spacing: {
      sm: '8px',
      md: '12px',
      lg: '16px',
      xl: '24px',
    }
  }
}
```

Here, we've overridden Tailwind's default spacing utilities and generated classes like `w-lg` and `h-md` instead.

Alternatively, we can add the "missing" utilities while keeping all the defaults as well:

```
module.exports = {
  theme: {
    extend: {
      spacing: {
        '13': '3.25rem',
        '15': '3.75rem',
        '128': '32rem',
        '144': '36rem',
      }
    }
  }
}
```

We can use these new utilities in the same way as default ones. For example, to apply our custom utilities for width and height, we write them like this: `w-15 h-13`.

Customizing Tailwind's Core Plugins

The last thing we can customize in the `theme` key is core plugins.

A **core plugin** is a utility with a series of different variations.

Here's the default definition for the `blur` plugin/utility:

```
blur: {
  0: '0',
```

```
    none: '0',
    sm: '4px',
    DEFAULT: '8px',
    md: '12px',
    lg: '16px',
    xl: '24px',
    '2xl': '40px',
    '3xl': '64px',
  }
```

The `blur` plugin applies a blur filter to an element. Each variation applies a different amount of blurring.

Let's see now how we can customize it. As with all utilities, we can either extend a plugin or override it.

In my view, the default blurring values are way too high. I prefer to have gentler blurring variations with a smooth transition between them. So here's an example of overriding a plugin's values:

```
module.exports = {
  theme: {
    blur: {
      'none': 'blur(0)',
      'sm': 'blur(2px)',
      DEFAULT: 'blur(4px)',
      'md': 'blur(6px)',
```

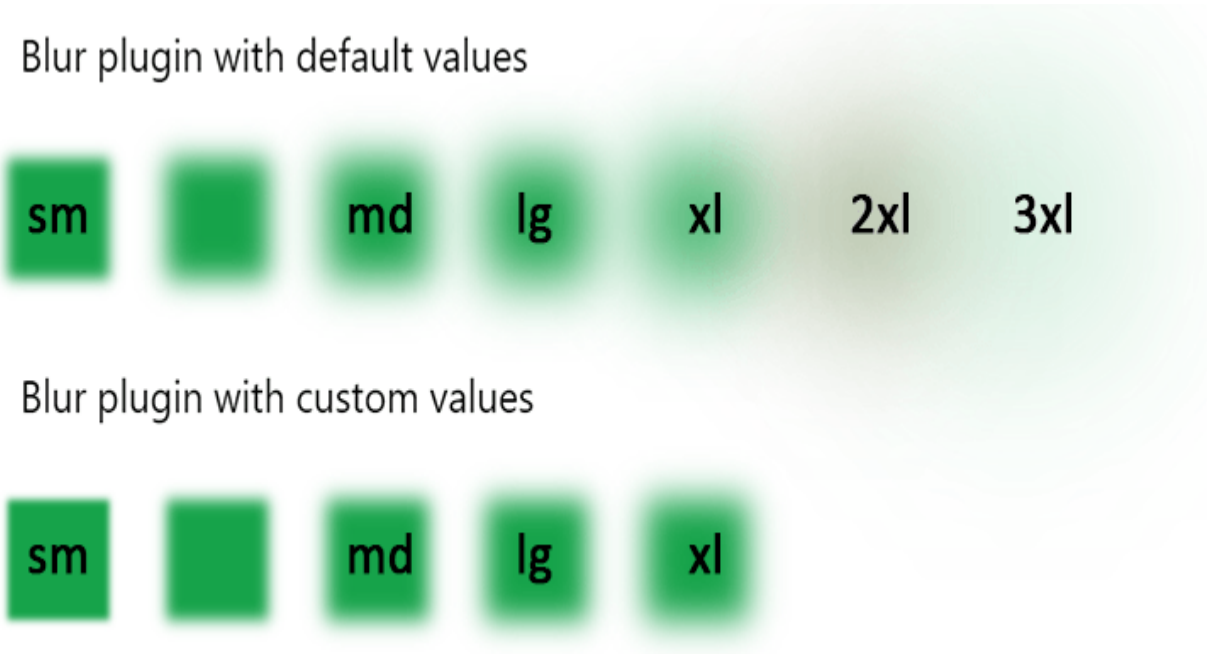
```
        'lg': 'blur(8px)',
        'xl': 'blur(10px)'
      }
    }
  }
```

We've added the plugin name as a key and provided our custom variations. The code above will produce the following classes: `blur-none`, `blur-sm`, `blur`, `blur-md`, and `blur-lg`.

DEFAULT KEY

It's a common convention to use a key of `DEFAULT` for the class without a suffix. This is supported by all core plugins.

I've removed the `2xl` and `3xl` variations because they're too exaggerated for me. The image below shows the difference.



The last two variations in the default version are barely visible. In my opinion, the customized version looks much better.

You can find instructions for customization of each core plugin at the end of each plugin's documentation page—like this one for the blur plugin.

It's also worth reading up on the default theme configuration for all core plugins.

A Practical Customization Example

In this section, we'll apply all we've learned so far. We'll replace the default responsive breakpoints with our own, we'll extend the theme

with additional colors and spacing utilities, and we'll add the font we included in our HTML file earlier to the default font stack.

Open the `tailwind.config.js` file and replace its content with the following:

```
// /tailwind.config.js
const colors = require('tailwindcss/colors')

module.exports = {
  content: ['./index.html'],
  theme: {
    screens: {
      'phone': '640px',
      'tablet': '768px',
      'laptop': '1024px',
      'desktop': '1280px',
    },
    extend: {
      colors: {
        primary: colors.yellow,
        secondary: colors.blue,
        neutral: colors.gray,
      },
      spacing: {
        '4.5': '1.125rem',
        '5.5': '1.375rem',
        '6.5': '1.625rem',
```

```
          '7.5': '1.875rem',
          '8.5': '2.125rem',
          '9.5': '2.375rem',
        },
        fontFamily: {
          'display': ['"Carter One"'],
        }
      }
    },
    plugins: [],
  }
```

Here, we've firstly added `index.html` to the `content` array.

Next, we've added four breakpoints with custom names that completely replace the default variants. The breakpoints are now more verbose but also more descriptive and easy to grasp.

Next, under the `extend` key, we've created custom named colors and used the Tailwind colors to define them. The reason here is similar. We want more descriptive names so they can be applied more intuitively—for example, when we create buttons.

Next, we've extended the spacing utilities with custom ones that give us a bit more precision. Sometimes our design needs to be pixel-perfect, so we might need a more precise scale.

Finally, we've extended the `fontFamily` core plugin to have a `display` font set. We've used only the Carter One font here, but we can add as many as we like. Using a custom font instead of the default fonts can make our design stand out. After all, we want our designs to be unique, right?

Now, to apply the changes, we should rebuild the styles manually—if we didn't run the build script with the `watch` flag. To do so, run `npm run dev:watch`.

We have the required styles, so now let's use them. Replace the header section in the `index.html` file with the following:

```
<!-- /index.html -->
<!doctype html>
<html>
  ...
<body>
  <header class="flex items-center justify-betwee
    <div class="flex-shrink-0 ml-6">
      <a href="#">
        <i class="fas fa-drafting-compass fa-2x t
        <span class="ml-1 text-3xl text-secondary
      </a>
    </div>
```

```
    <button id="nav-toggle" class="tablet:hidden
      <i class="fas fa-bars fa-2x"></i>
    </button>

    <div class="pl-6 w-full tablet:w-auto hidden
      <ul class="tablet:flex">
        <li class="mr-5.5 p-1 tablet:border-b-2 b
          <a class="text-secondary-200 cursor-def
        </li>
        <li class="mr-5.5 p-1">
          <a class="text-white hover:text-seconda
        </li>
        <li class="mr-5.5 p-1">
          <a class="text-white hover:text-seconda
        </li>
        <li class="mr-5.5 p-1">
          <a class="text-white hover:text-seconda
        </li>
        <li class="mr-5.5 p-1">
          <a class="text-white hover:text-seconda
        </li>
        <li class="mr-5.5 p-1">
          <a class="text-white hover:text-seconda
        </li>
      </ul>
    </div>
  </header>
```

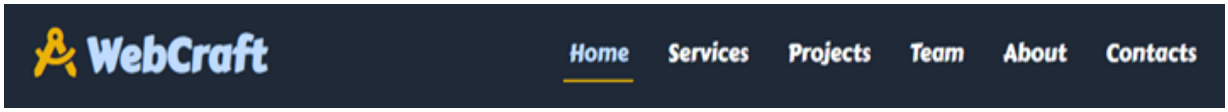
```
</body>
</html>
```

What we've just done here is replace the following classes:

- all `md:` occurrences with `tablet:`
- `yellow`, `blue`, and `gray` classes with `primary`, `secondary`, and `neutral` respectively
- `mr-6` with `mr-5.5`

We've also added a `font-display` class in the header.

The image below shows the final result after the changes.



Of course, visually the most notable difference is the new font. The other changes (apart from the margin tweaks) are just class-name replacements that affect the visual appearance, although the code is now a bit more descriptive.

Adding Base Classes

As we already know, Tailwind automatically adds Preflight base styles to each project by default. These settings do things like remove the

default browser styles for the headings, paragraphs, lists, and so on, which may be unwanted.

So if we don't want Preflight styles, we can disable them by setting the `preflight` value to false:

```
module.exports = {
  corePlugins: {
    preflight: false,
  }
}
```

Usually this isn't necessary, but it's still worth knowing it's an option.

In either case, with Preflight classes or without them, we can add our own base classes that can override or extend Preflight, depending on whether a class already exists or not.

In the next example, we'll override the classes for `<h1>`, `<h2>`, and `<p>` elements.

Open `styles.css` and add the following:

```
/* /styles.css */
@tailwind base;
@tailwind components;
```

```
@tailwind utilities;

@layer base {
  h1 {
    @apply text-2xl;
  }
  h2 {
    @apply text-xl;
  }
  h1, h2, p {
    @apply my-6 mx-4;
  }
}
```

Here, we've changed the size for some headings and added margins for the same headings and all paragraphs.

REBUILDING STYLES

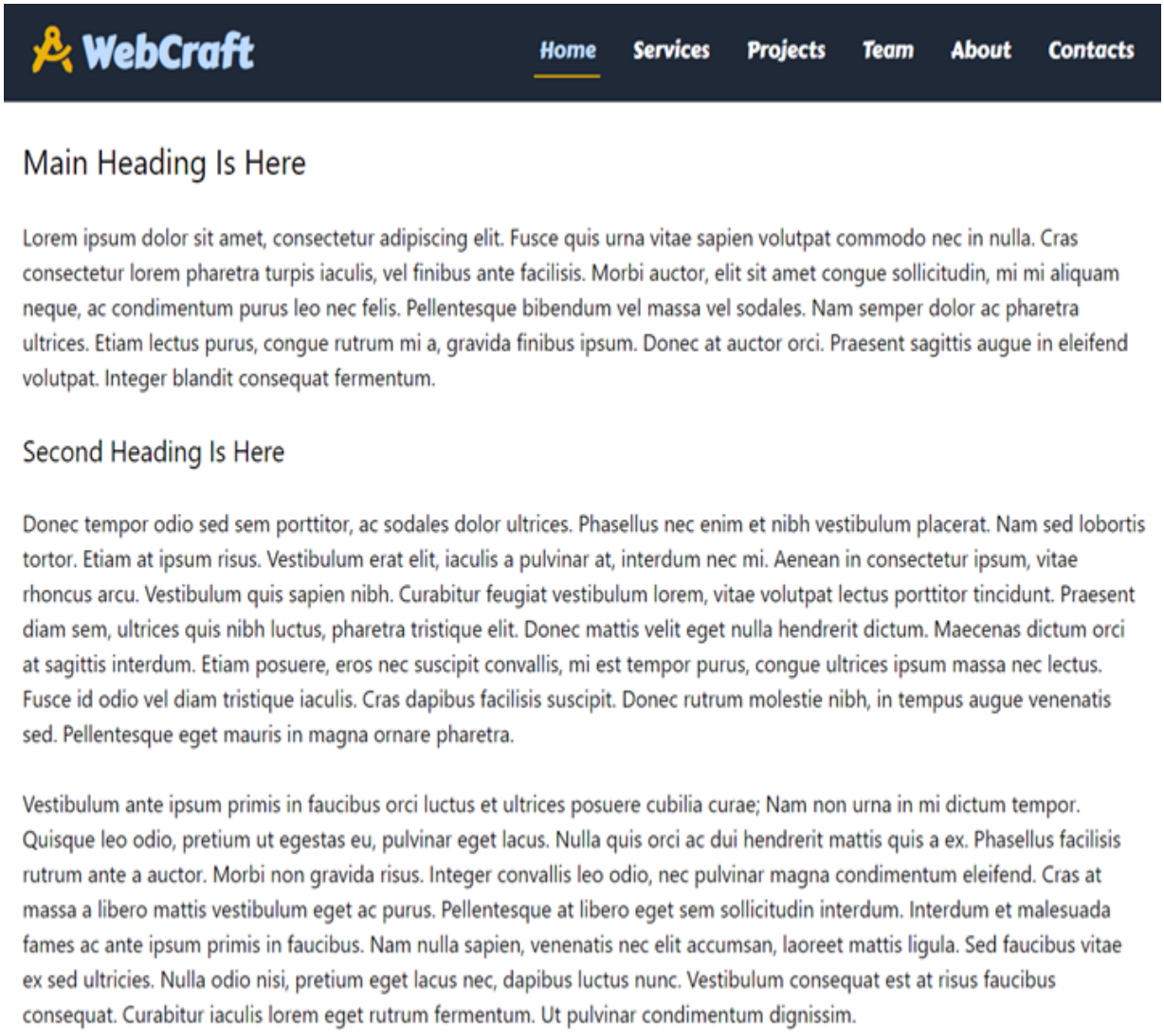
You should rebuild the styles to apply the changes. To do so, run the `npm run dev:watch` command.

To see this in action, open `index.html` and add this content below the header:

```
<!-- /index.html -->
<!doctype html>
<html>
```

```
<body>
  <h1>Main Heading Is Here</h1>


  <p>Lorem ipsum dolor sit amet, consectetur adip

  <h2>Second Heading Is Here</h2>

  <p>Donec tempor odio sed sem porttitor, ac soda

  <p>Vestibulum ante ipsum primis in faucibus orc
</body>
</html>
```

Now, when you open it, you should see the headings and paragraphs displayed with a space between them.



Creating Configuration Presets

If we want to reuse our configuration across different projects, Tailwind offers us a way to do so by creating reusable configuration presets. A **configuration preset** is a number of settings defined in the exact same way as those from the `tailwind.config.js` file. The only difference is that they're put in a different, separate file.

Using presets is very useful for creating branding and/or design systems.

Let's suppose we have brand colors that we want to use in a project or perhaps in multiple projects. In this scenario, we can create a preset with the brand colors. Let's create a `brand-colors-preset.js` file and put the following content inside:

```
// /brand-colors-preset.js
const colors = require('tailwindcss/colors')

module.exports = {
  theme: {
    extend: {
      colors: {
        primary: colors.yellow,
        secondary: colors.blue,
        neutral: colors.gray,
      }
    }
  }
}
```

Here, we've moved the colors from the main configuration file into the preset. To include the preset into your main configuration, add it under the `presets` key, as in the example below:

```
// /tailwind.config.js
module.exports = {
  content: ['./index.html'],
  presets: [
    require('./brand-colors-preset.js')
  ],
  theme: {
    screens: {
      'phone': '640px',
      'tablet': '768px',
      'laptop': '1024px',
      'desktop': '1280px',
    },
    extend: {
      spacing: {
        '4.5': '1.125rem',
        '5.5': '1.375rem',
        '6.5': '1.625rem',
        '7.5': '1.875rem',
        '8.5': '2.125rem',
        '9.5': '2.375rem',
      },
      fontFamily: {
        'display': ['"Carter One"'],
      }
    }
  },
```

```
    plugins: [],
  }
```

Here, we've removed the colors from the `colors` key and instead added them as a preset defining our brand colors. This gives us more flexibility to easily change the brand colors in future or use completely different colors if we wish.

REBUILDING STYLES

You should rebuild the styles to apply the changes. To do so, run the `npm run dev:watch` command.

MERGING

Just as the `tailwind.config.js` settings are merged with the default configuration, the preset settings are merged with the `tailwind.config.js`. You can learn more about merging in the Tailwind documentation.

We can also use multiple presets:

```
module.exports = {
  presets: [
    require('responsive-breakpoints-preset.js'),
    require('brand-colors-preset.js'),
```

```
      require('brand-fonts-preset.js'),
    ]
  }
```

If there are overlapping classes in two or more presets, the classes specified in the last preset will take precedence.

Conclusion

In this chapter, we explored various ways to configure Tailwind, such as tweaking the default theme by overriding and/or extending it, creating reusable presets, and tweaking the base Tailwind styles.

In the next chapter, we'll end this book by exploring the use of plugins, making custom plugins, and building a custom design system.

Chapter 5: Working with Tailwind Plugins

In the last chapter, we learned the most important ways to customize and extend a default Tailwind theme.

Now, in this final chapter, we'll end our journey by exploring the one of Tailwind's most powerful features: plugins. We'll firstly look at how to use the official Tailwind plugins. Then we'll learn how to create our own custom Tailwind plugins.

Getting Started

Let's prepare for this tutorial by creating a new project (a process we covered in the Chapter 2).

PROJECT CODE

You can find the finished project for this chapter in the code repo for this book.

Once we have our new project ready, the next step is to modify the `content` key, inside `tailwind.config.js`, like so:

```
// /tailwind.config.js
module.exports = {
  content: ['./examples/*.html'],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

We'll create several HTML example files before the end of this tutorial, so this tells Tailwind where to look for them to build the required styles.

Using Official Tailwind Plugins

In this section, we'll examine the Typography plugin—which is one of four plugins officially maintained by the the Tailwind team.

The other three are:

- Forms, which adds minimal default styles to all basic form elements
- Aspect Ratio, which adds utilities for declaring a fixed aspect ratio to elements
- Line Clamp, which adds utilities for text truncating after a fixed number of lines

The Typography plugin adds some default typographic styles that are difficult or impossible to add manually. For example, a post content in a post template can be included in Markdown format, which will be converted and rendered as HTML, but we don't have access to that HTML in the template:

```
<!-- A post template -->
<article>
  {{ markdown }}
</article>
```

In a situation like this, the Typography plugin is used to inject the required classes dynamically in the rendered HTML.

To use the plugin, we need to install it first:

```
npm install -D @tailwindcss/typography
```

Then, add the plugin inside the `tailwind.config.js` file, in the `plugins` array:

```
// /tailwind.config.js
module.exports = {
  // ...
  plugins: [
```

```
    require('@tailwindcss/typography')
  ]
}
```

Now, create new a `examples` directory and add a `typography.html` file in it with the following content:

```
<!-- /examples/typography.html -->
<!doctype html>
<html>
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-wid
  <title>Tailwind Typography Plugin Example</titl
  <link href="../tailwind.css" rel="stylesheet">
</head>
<body>
  <div class="bg-sky-50">
    <article>
      <h1>Some Nice Title Here</h1>

      <p class="lead">Lorem ipsum dolor sit amet,
      <h2>1. A Heading 2 Here</h2>
      <blockquote><p>Lorem ipsum dolor sit amet,
      <p>Aliquam id nulla dignissim felis bibendu
      <figure>
        <img src="https://source.unsplash.com/ZEi
        <figcaption>Donec blandit pulvinar leo et
      </figure>
```

```
        </figure>
        <h2>2. Another Heading 2 Here</h2>
        <p>Vestibulum congue felis at posuere commo
        <ul>
          <li>List item</li>
          <li>List item</li>
          <li>List item</li>
        </ul>
        <p>Aenean scelerisque urna id dictum tempor
        <ol>
          <li>First item</li>
          <li>Second item</li>
          <li>Third item</li>
        </ol>
        <p>Pellentesque tincidunt non orci id congu
      </article>
    </div>
  </body>
  </html>
```

Here, we have a bare-bones article structure with most of the elements the plugin can style typographically. I only added a light blue background to the `<div>` container to make the screenshots more distinguishable. The `lead` class, in the first paragraph, will be used by the Typography plugin later on.

Run `npm run dev:watch` and open the file in the browser. The image below shows what we should see.

Some Nice Title Here

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas varius vitae ipsum et commodo. In scelerisque est magna, ut fringilla purus congue eu. Mauris id metus ac metus porta aliquet. Aliquam quam ipsum, consequat malesuada lectus nec, blandit condimentum enim.

A Heading 2 Here

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas varius vitae ipsum et commodo.
Aliquam id nulla dignissim felis bibendum aliquam. Cras vulputate blandit semper. Nam quam dolor, tincidunt non odio ac, condimentum molestie justo. In ornare maximus tortor, aliquam consequat arcu sagittis id.

Donec blandit pulvinar leo et tincidunt.

Another Heading 2 Here

Vestibulum congue felis at posuere commodo. Praesent sapien magna, aliquet ut efficitur et, luctus at neque. Donec vitae nunc convallis, maximus ex sit amet, consequat tellus. Aenean eleifend cursus urna, sed fermentum felis cursus eget. Sed accumsan hendrerit turpis at ullamcorper. Integer quam sapien, rutrum ac pharetra eget, maximus id lacus.

- List item
- List item
- List item

Aenean scelerisque urna id dictum tempor. Pellentesque ipsum orci, convallis eget purus nec, placerat laoreet nulla. Nullam vitae lectus porta, lacinia neque at, rutrum felis.

1. First item
2. Second item
3. Third item

Pellentesque tincidunt non orci id congue. Donec blandit pulvinar leo et tincidunt. Sed venenatis venenatis justo, ut congue neque lobortis sit amet. Nam tempus vehicula nisi, vitae commodo magna condimentum id.

At this stage, the article is mostly unstyled. Now it's time to see the magic of the Typography plugin. To use it in HTML, we need to include its classes and to utter the magic word: *prose*.

`prose` is the base class used by the Typography plugin. All other classes also start with this class, as we'll see shortly.

Add the `prose` class to the article tag like so: `<article class="prose">`. Now reload the page and behold the magic, as pictured below.

Some Nice Title Here

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas varius vitae ipsum et commodo. In scelerisque est magna, ut fringilla purus congue eu. Mauris id metus ac metus porta aliquet. Aliquam quam ipsum, consequat malesuada lectus nec, blandit condimentum enim.

A Heading 2 Here

> ***"Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas varius vitae ipsum et commodo."***

Aliquam id nulla dignissim felis bibendum aliquam. Cras vulputate blandit semper. Nam quam dolor, tincidunt non odio ac, condimentum molestie justo. In ornare maximus tortor, aliquam consequat arcu sagittis id.

Donec blandit pulvinar leo et tincidunt.

Another Heading 2 Here

Vestibulum congue felis at posuere commodo. Praesent sapien magna, aliquet ut efficitur et, luctus at neque. Donec vitae nunc convallis, maximus ex sit amet, consequat tellus. Aenean eleifend cursus urna, sed fermentum felis cursus eget. Sed accumsan hendrerit turpis at ullamcorper. Integer quam sapien, rutrum ac pharetra eget, maximus id lacus.

- List item
- List item
- List item

Aenean scelerisque urna id dictum tempor. Pellentesque ipsum orci, convallis eget purus nec, placerat laoreet nulla. Nullam vitae lectus porta, lacinia neque at, rutrum felis.

1. First item
2. Second item
3. Third item

Pellentesque tincidunt non orci id congue. Donec blandit pulvinar leo et tincidunt. Sed venenatis venenatis justo, ut congue neque lobortis sit amet. Nam tempus vehicula nisi, vitae commodo magna condimentum id.

The article looks much better with the plugin's styles applied. It's not perfect, but it's a great foundation for further customization.

Now let's try the Typography plugin's dark mode. First, replace the `bg-sky-50` class of the container `<div>` with `bg-sky-900` to create a dark background. Then add the `dark:prose-invert` class to the article like so: `<article class="prose dark:prose-invert">`.

Reload the page, and boom. The image below shows what we get now.

Some Nice Title Here

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas varius vitae ipsum et commodo. In scelerisque est magna, ut fringilla purus congue eu. Mauris id metus ac metus porta aliquet. Aliquam quam ipsum, consequat malesuada lectus nec, blandit condimentum enim.

A Heading 2 Here

> ***"Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas varius vitae ipsum et commodo."***

Aliquam id nulla dignissim felis bibendum aliquam. Cras vulputate blandit semper. Nam quam dolor, tincidunt non odio ac, condimentum molestie justo. In ornare maximus tortor, aliquam consequat arcu sagittis id.

Donec blandit pulvinar leo et tincidunt.

Another Heading 2 Here

Vestibulum congue felis at posuere commodo. Praesent sapien magna, aliquet ut efficitur et, luctus at neque. Donec vitae nunc convallis, maximus ex sit amet, consequat tellus. Aenean eleifend cursus urna, sed fermentum felis cursus eget. Sed accumsan hendrerit turpis at ullamcorper. Integer quam sapien, rutrum ac pharetra eget, maximus id lacus.

- List item
- List item
- List item

Aenean scelerisque urna id dictum tempor. Pellentesque ipsum orci, convallis eget purus nec, placerat laoreet nulla. Nullam vitae lectus porta, lacinia neque at, rutrum felis.

1. First item
2. Second item
3. Third item

Pellentesque tincidunt non orci id congue. Donec blandit pulvinar leo et tincidunt. Sed venenatis venenatis justo, ut congue neque lobortis sit amet. Nam tempus vehicula nisi, vitae commodo magna condimentum id.

This little switch was just for showing you that the dark side exists, in case you need it! Let's now go back to the default version and continue our journey with the light theme. Remove the `dark:prose-invert` class and set the container background back to `bg-sky-50`.

The real power of the Typography plugin is that it provides modifiers for the typographic elements so you can customize them individually.

Let's try some of them now. Add the following classes to the `<article>` tag:

```
<!-- /examples/typography.html -->
<!-- ... -->
<body>
  <div class="bg-sky-50 p-6">
    <article class="prose
    prose-h1:underline prose-h1:underline-offset-
    prose-h2:first-letter:text-cyan-600
    prose-headings:text-cyan-900
    prose-lead:text-cyan-600
    prose-p:first-line:italic
    prose-blockquote:text-cyan-600 prose-blockquo
    prose-figure:mx-6
    prose-figcaption:text-center
    prose-img:rounded-lg prose-img:drop-shadow-lg
    prose-li:marker:text-cyan-600
```

```
      ">
        <h1>Some Nice Title Here</h1>
        <!-- ... -->
```

Don't Forget `prose`

The base `prose` class must always be present before using the other classes.

Here, I've added some space around the article (for a nicer look and feel) by adding some padding (`p-6`) to the container `<div>` .

I've grouped the `prose` classes by element (for better readability and maintainability), so each line contains classes for one specific element.

To set a utility for a particular element, we start with the `prose` class, followed by a colon and then the utility: `prose-h1:underline` .

Only one utility can be added for a `prose` class instance. We can't add a sequence of utilities like this: `prose-h1:underline underline-offset-8` .

Instead, we must define them individually: `prose-h1:underline prose-h1:underline-offset-8` .

When we stack `prose` classes with other modifiers, we start with the `prose` class, followed by the modifier(s), and lastly the utility, like so: `prose-h2:first-letter:text-cyan-600`.

`prose-lead` targets an element with class `lead` as the first paragraph of the article.

Basically, these are the rules for using the `prose` classes.

Let's go back to the browser now to see what we've achieved so far. Reload the page and enjoy a fully styled article, as shown below.

Some Nice Title Here

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas varius vitae ipsum et commodo. In scelerisque est magna, ut fringilla purus congue eu. Mauris id metus ac metus porta aliquet. Aliquam quam ipsum, consequat malesuada lectus nec, blandit condimentum enim.

1. A Heading 2 Here

> ***"Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas varius vitae ipsum et commodo."***

Aliquam id nulla dignissim felis bibendum aliquam. Cras vulputate blandit semper. Nam quam dolor, tincidunt non odio ac, condimentum molestie justo. In ornare maximus tortor, aliquam consequat arcu sagittis id.

Donec blandit pulvinar leo et tincidunt.

2. Another Heading 2 Here

Vestibulum congue felis at posuere commodo. Praesent sapien magna, aliquet ut efficitur et, luctus at neque. Donec vitae nunc convallis, maximus ex sit amet, consequat tellus. Aenean eleifend cursus urna, sed fermentum felis cursus eget. Sed accumsan hendrerit turpis at ullamcorper. Integer quam sapien, rutrum ac pharetra eget, maximus id lacus.

- List item
- List item
- List item

Aenean scelerisque urna id dictum tempor. Pellentesque ipsum orci, convallis eget purus nec, placerat laoreet nulla. Nullam vitae lectus porta, lacinia neque at, rutrum felis.

1. First item
2. Second item
3. Third item

Pellentesque tincidunt non orci id congue. Donec blandit pulvinar leo et tincidunt. Sed venenatis venenatis justo, ut congue neque lobortis sit amet. Nam tempus vehicula nisi, vitae commodo magna condimentum id.

Building Custom Tailwind Plugins

In this section, we'll look at how to create custom Tailwind plugins.

We'll create two small plugins:

- a `counters` plugin, which will add the ability to automatically add numbers to the document headings or whatever other elements we want
- an `arrows` plugin, which will add the ability to incorporate CSS arrow shapes/icons into our designs

---

RUNNING `DEV:WATCH`

From now on, we don't need to stop and run the `dev:watch` command again, because the custom plugins don't need to be installed. But it's good to make sure the command is still running before we test the HTML example files. Sometimes, while we're making changes, a syntactic error can stop the execution of the script. In this case, we'll see an appropriate error message in the terminal, such as `SyntaxError: Unexpected identifier`. To apply the changes we've made and see a proper HTML result, we need to start the script again with `npm run dev:watch`.

Creating the Counters Plugin

This plugin will take advantage of the CSS counters feature. It will add automatic numbers for document headings by default. Also, it will allow for automatic numbering of any other elements to be added by using the necessary classes provided by the plugin.

To start, create a new `plugins` directory and add a new `counters.js` file in it with the following content:

```
// /plugins/counters.js
const plugin = require('tailwindcss/plugin')

const counters = plugin(function({ addBase, addC
  // put plugin logic here
})

module.exports = counters
```

A plugin is created by the `plugin()` function (required in the beginning), which takes an anonymous function as its first argument. The anonymous function takes a single object as argument, which we can destructure for convenience. The destructured properties are Tailwind

functions for customizing various layers of Tailwind's default styles. Here's a list of each available function:

- `addBase()` adds base styles
- `addComponents()` adds static component styles
- `matchComponents()` adds dynamic component styles
- `addUtilities()` adds static utility styles
- `matchUtilities()` adds dynamic utility styles
- `addVariant()` adds custom variants
- `theme()` provides access to values in the user's theme configuration
- `config()` provides access to values in the user's Tailwind configuration
- `corePlugins()` checks if a core plugin is enabled
- `e()` manually escapes strings meant to be used in class names

Now, add the following `addBase()` function:

```
// /plugins/counters.js
const plugin = require('tailwindcss/plugin')

const counters = plugin(function({ addBase, theme
  addBase({
    'h1': {
      counterReset: 'level-1'
    },
```

```
      'h2': {
        counterReset: 'level-2'
      },
      'h3': {
        counterReset: 'level-3'
      },
      'h2::before, h3::before, h4::before': {
        color: theme('colors.slate.600')
      },
      'h2::before': {
        counterIncrement: 'level-1',
        content: 'counter(level-1) ". "'
      },
      'h3::before': {
        counterIncrement: 'level-2',
        content: 'counter(level-1) "." counter(leve
      },
      'h4::before': {
        counterIncrement: 'level-3',
        content: 'counter(level-1) "." counter(leve
      },
    })
  })

  module.exports = counters
```

Here, we've created three counters by using the `counterReset` property.

Then we've set the counters to be used for `h2`, `h3`, and `h4` heading elements by using the `counterIncrement` property. We've set the actual numbers by using the `content` property, where we've used the `counter()` function—which returns the current value of the named counter. We've also concatenated each nested heading with the previous one(s).

Finally, we've added a color for the numbers in the headings by using the `theme()` function, which allows us to reuse the default Tailwind colors. Note that the dash we usually see between color name and number (such as `slate-600`) is replaced by a dot here: `slate.600`. This is because the colors for the Tailwind theme are represented by JavaScript objects, so we need to use dot notation to access their properties and methods.

Let's test what we've achieved so far. To do so, in the `examples` directory, create a new `counters.html` file with the following content:

```
<!-- /examples/counters.html -->
<!DOCTYPE html>
<html lang="en">
<head>
```

```
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=
    <meta name="viewport" content="width=device-wid
    <title>CSS Counters Example</title>

    <link href="../tailwind.css" rel="stylesheet">
  </head>
  <body class="prose">
    <div class="m-3">
      <p class="text-3xl text-red-700">Counters Bas
      <h1>Web Development Languages</h1>
        <h2>HTML</h2>
        <h2>CSS</h2>
          <h3>Tailwind</h3>
          <h3>Bootstrap</h3>
        <h2>JavaScript</h2>
          <h3>Node</h3>
            <h4>Express</h4>
          <h3>Vue</h3>
            <h4>Vuetify</h4>
            <h4>Nuxt</h4>
    </div>
  </body>
  </html>
```

Here, to make things more readable, we've used the `prose` class from the Typography plugin.

Now, add the plugin to the Tailwind configuration:

```
// /tailwind.config.js
module.exports = {
  // ...
  plugins: [
    require('@tailwindcss/typography'),
    require('./plugins/counters')
  ]
}
```

THE `.JS` EXTENSION

There's no need to add the `.js` extension. Tailwind automatically recognizes the file.

Finally, open the `counters.html` in your browser. The image below shows what you should see.

Counters Based on Heading Elements:

Web Development Languages

1. HTML

2. CSS

2.1 Tailwind

2.2 Bootstrap

3. JavaScript

3.1 Node

3.1.1 Express

3.2 Vue

3.2.1 Vuetify

3.2.2 Nuxt

As you can see, with no classes added to the HTML, we get nice and proper numbering for the heading elements.

Great! But what if we want to add numbering for different elements? We can do this too. Let's see how now.

Go to the `counters.js` file and replace its content with the following:

```
// /plugins/counters.js
const plugin = require('tailwindcss/plugin')

const counters = plugin(function({ addBase, addU
  addBase({
    'h1': {
      counterReset: 'level-1'
    },
    'h2': {
      counterReset: 'level-2'
    },
    'h3': {
      counterReset: 'level-3'
    },
    'h2::before, h3::before, h4::before': {
      color: theme('colors.slate.600')
    },
    'h2::before': {
```

```
        counterIncrement: 'level-1',
        content: 'counter(level-1) ". "'
      },
      'h3::before': {
        counterIncrement: 'level-2',
        content: 'counter(level-1) "." counter(leve
      },
      'h4::before': {
        counterIncrement: 'level-3',
        content: 'counter(level-1) "." counter(leve
      },
    })
    addUtilities({
      '.collection': {
        counterReset: 'collection'
      },
      '.item::before': {
        counterIncrement: 'collection',
        content: 'counters(collection,".") " "'

      }
    })
  })

  module.exports = counters
```

Here, we've added the `addUtilities()` function (which we first define in the destructured object) to create two classes. The first—the

`collection` class—creates a new counter. The second—the `item` class—adds a number before the element on which it's used. Here, we've used the `counters()` function (ending with `s`), which returns a concatenated string representing the current values of the named counters.

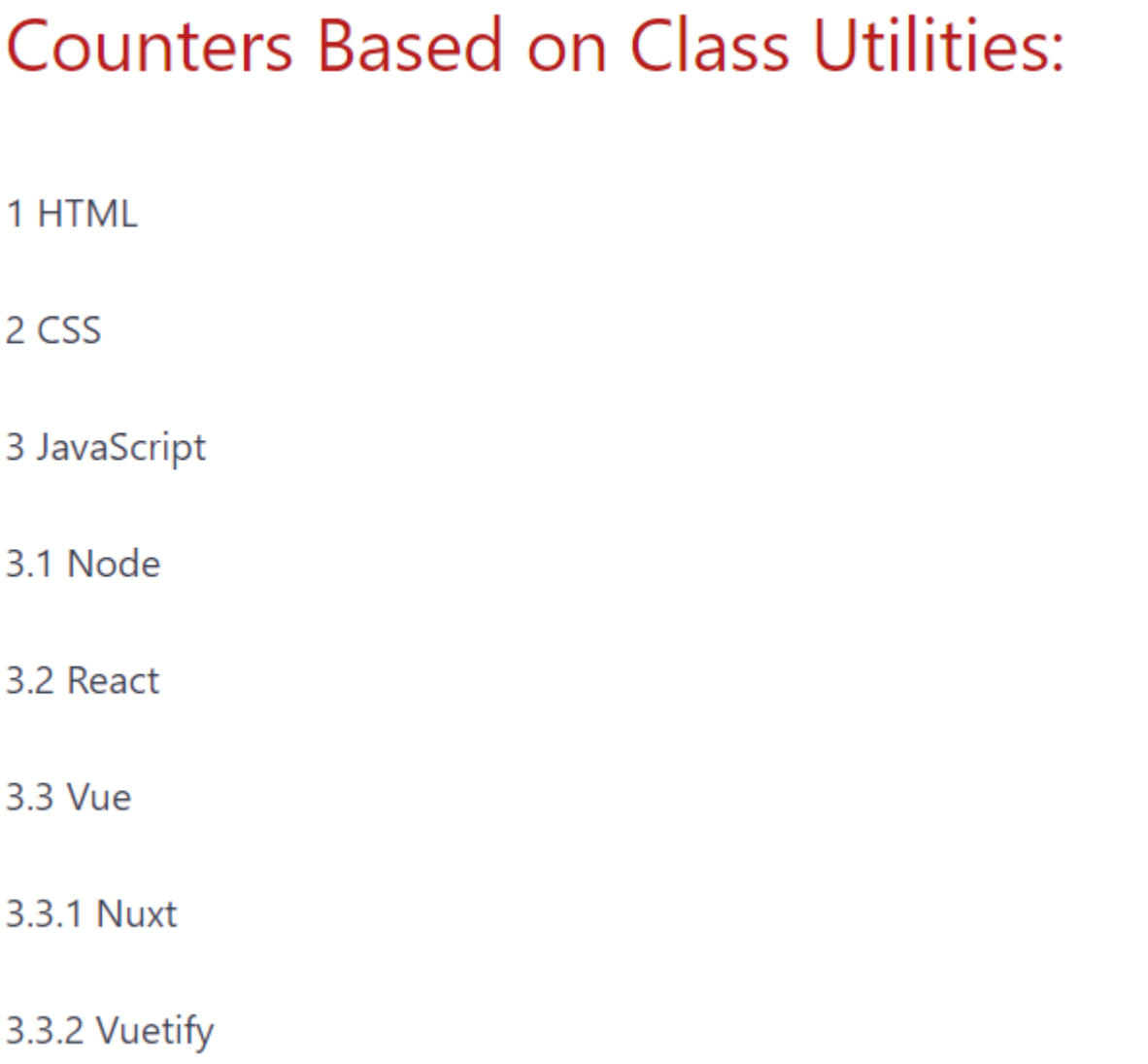
To test this feature, replace the content of `counters.html` with the following:

```
<!-- /examples/counters.html -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=
  <meta name="viewport" content="width=device-wid
  <title>CSS Counters Example</title>

  <link href="../tailwind.css" rel="stylesheet">
</head>
<body class="prose">
  <div class="m-3">
    <p class="text-3xl text-red-700">Counters Bas
    <h1>Web Development Languages</h1>
      <h2>HTML</h2>
      <h2>CSS</h2>
        <h3>Tailwind</h3>
        <h3>Bootstrap</h3>
```

```
      <h2>JavaScript</h2>
        <h3>Node</h3>
          <h4>Express</h4>
        <h3>Vue</h3>
          <h4>Vuetify</h4>
          <h4>Nuxt</h4>

    <p class="text-3xl text-red-700 mt-12">Counte
    <div class="collection">
      <p class="item">HTML</p>
      <p class="item">CSS</p>
      <p class="item">JavaScript</p>
      <div class="collection">
        <p class="item">Node</p>
        <p class="item">React</p>
        <p class="item">Vue</p>
        <div class="collection">
          <p class="item">Nuxt</p>

          <p class="item">Vuetify</p>
        </div>
      </div>
    </div>
  </div>
</body>
</html>
```

Here, we've added a section—below the first example—in which we're using the plugin classes. By nesting our collections with items, we get nested numbering for the paragraph elements.

Open or reload the `counters.html` file. The image below shows what you should see.



Because of the utilities we've added, we can add numbering to any element we want.

Creating the Arrows Plugin

This arrows plugin will use a technique for creating different shapes with CSS—which shouldn't be confused with the CSS Shapes module). It will produce four arrows that we can use as icons in our designs.

To start, create a new `arrows.js` file in the `plugins` directory with the following content:

```
// /plugins/arrows.js
const plugin = require('tailwindcss/plugin')

const arrows = plugin(function({ addComponents }
  addComponents({
    '.arrow': {
      border: 'solid black',
      borderWidth: '0 3px 3px 0',
      display: 'inline-block',
      padding: '3px',
      marginLeft: '5px'
    },
    '.arrow-up': {
```

```
      transform: 'rotate(-135deg)'
    },
    '.arrow-right': {
      transform: 'rotate(-45deg)'
    },
    '.arrow-down': {
      transform: 'rotate(45deg)'
    },
    '.arrow-left': {
      transform: 'rotate(135deg)'
    },
  })
})

module.exports = arrows
```

Here, we've used the `addComponents()` function to define the necessary classes. The first `arrow` class adds the base for all arrows. Then we've added the remaining four classes that define the four possible directions of an arrow.

To test it, create a new `arrows.html` file in the `examples` directory with the following content:

```
<!-- /examples/arrows.html -->
<!DOCTYPE html>
<html lang="en">
```

```
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=
  <meta name="viewport" content="width=device-wid
  <title>CSS Arrows Example</title>
  <link href="../tailwind.css" rel="stylesheet">
</head>
<body class="prose p-6">
  <h2>Tree-View Dynamic List with CSS Arrows:</h2
  <ul>
    <li><span class="toggle cursor-pointer after:
      <ul class="section hidden">
        <li>Node</li>
        <li>React</li>
        <li><span class="toggle cursor-pointer af
          <ul class="section hidden">
            <li>Nuxt</li>
            <li>Vuetify</li>
            <li><span class="toggle cursor-pointe
              <ul class="section hidden">
                <li>Components</li>
                <li>Composables</li>
                <li>Directives</li>
              </ul>
            </li>
          </ul>
        </li>
      </ul>
    </li>
```

```
      </li>
    </ul>


    <script>
      let toggles = document.getElementsByClassName
      let sections = document.getElementsByClassNam

      for (let i = 0; i < toggles.length; i++) {
        toggles[i].addEventListener("click", functi
          let section = sections[i]
          section.classList.contains('hidden') ? se
          this.classList.toggle("after:arrow-down")
        })
      }
    </script>
  </body>
  </html>
```

USING `PROSE` FOR READABILITY

The `prose` class is used again in this example for better readability.

Here, we've created a dynamic tree-view list. Each list item (marked with a `toggle` class), which contains a nested list or lists (marked with a `section` class), gets a right arrow icon by using the `arrow`

and `arrow-right` classes and `after:` pseudo-element. Each nested list is hidden by default with the help of the `hidden` class.

In the `<script>` tag, we first get all toggles and sections. Then we iterate through toggles and add a click event listener for each one. When the function is executed, it toggles the `hidden` and `after:arrow-down` classes. This results in showing/hiding the corresponding nested list(s) and changes the toggle icon accordingly.

Let's add the plugin to the Tailwind configuration:

```
// /tailwind.config.js
module.exports = {
  // ...
  plugins: [
    require('@tailwindcss/typography'),
    require('./plugins/counters'),
    require('./plugins/arrows')
  ]
}
```

Now, open `arrows.html` in the browser. In the screenshot below, you can see how the list should look and behave.

Tree-View Dynamic List with CSS Arrows:

- JavaScript ⌄
  - Node
  - React
  - Vue ⌄
    - Nuxt
    - Vuetify
    - Quasar ❯

As you can see, the arrows help to make the tree-view list more intuitive and descriptive.

Congratulations! You've just created two useful custom plugins.

Finding Community Plugins

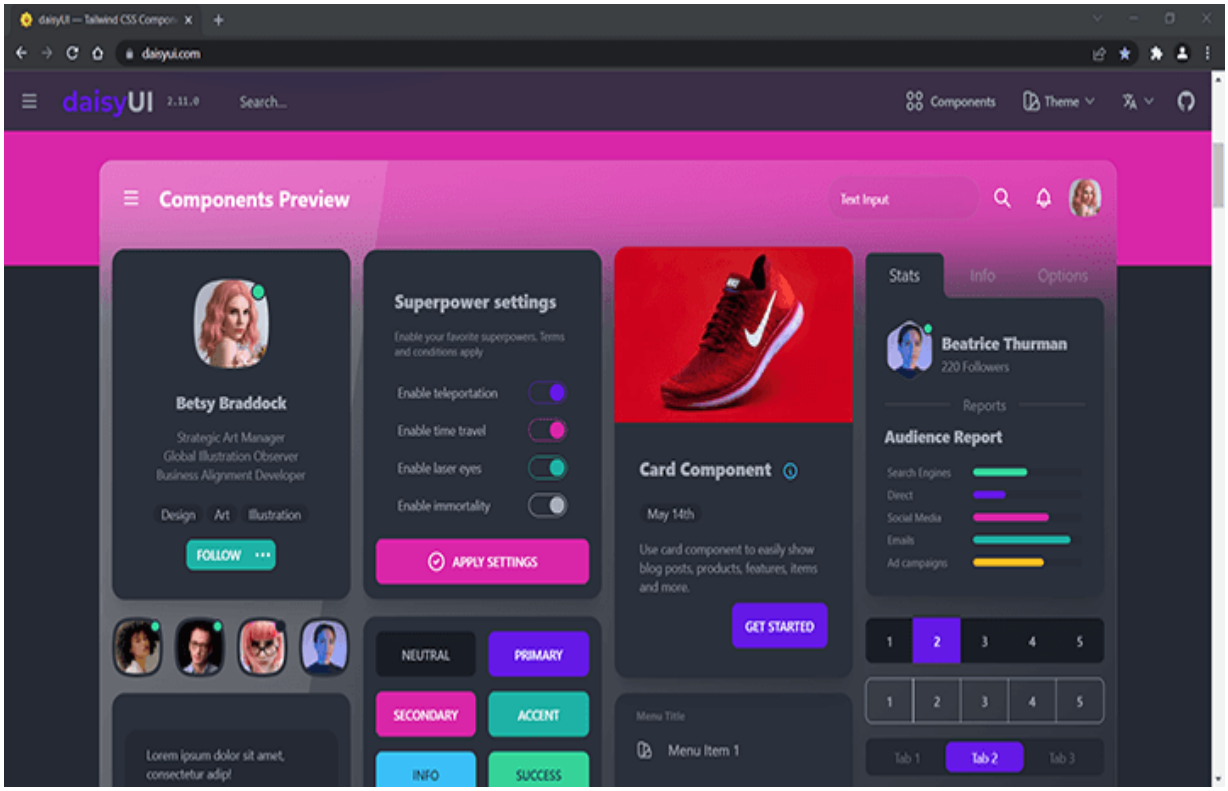
Building your own plugin is great indeed, but it's always a good idea to firstly check if there's already an existing plugin offering the functionality you need. If such a plugin exists, it's probably a better option to use that than to reinvent the wheel.

The best place to check for existing plugins and other cool stuff is Awesome Tailwind CSS.

Another one is Tailwind Toolbox.

When you choose a plugin, check if it's still maintained, and also which Tailwind version it's written for. If it's popular and widely used, this can be huge bonus too.

I want to recommend one plugin in particular that can be very useful: daisyUI, by Pouya Saadeghi.



daisyUI provides a rich set of highly customizable and themeable components that come in two versions: styled and unstyled. This is pretty flexible and means we can adapt it to any project.

You may be wondering why we don't just use Bootstrap components instead (or any other component-based framework). Well, the difference is that daisyUI offers a much easier way to customize the components to suit our needs. daisyUI is fully themeable and offers many pre-made themes out of the box. It even has a theme generator for building our own themes. So, give it a try.

There's a full list of the components provided by daisyUI on the daisyUI site.

Conclusion

In this chapter, we firstly looked at how to use the official Typography plugin to add beautiful default styling to the most used typographic elements. We then got our hands dirty by creating two useful custom plugins.

And this marks the end of this book. Phew, it's been a pretty long ride. If you've made it through the entire book, congratulations! Let's summarize what's been achieved:

1. In the first part, we learned how to create a simple website design by using the basic Tailwind utilities.
2. In the second part, we ventured a bit deeper and learned how to create Tailwind components.
3. In the third part, we learned even more advanced Tailwind capabilities such as building grid layouts, creating typographic styles, and using filters, transforms and transitions.
4. In the forth part, we explored the core of Tailwind theme customization and extension.
5. In this fifth and final part, we learned about the last Tailwind customization weapon: plugins. We learned both how to create custom plugins and how to use the existing ones.

Now you can consider yourself to be an advanced Tailwind user. You have the complete skills package necessary to build any design you can imagine. The sky's the limit.

Thanks for traveling with me through this book. Best of luck in your future journey with Tailwind!