Core Concepts

Adding Custom Styles

Best practices for adding your own custom styles to Tailwind.

Often the biggest challenge when working with a framework is figuring out what you're supposed to do when there's something you need that the framework doesn't handle for you.

Tailwind has been designed from the ground up to be extensible and customizable, so that no matter what you're building you never feel like you're fighting the framework.

This guide covers topics like customizing your design tokens, how to break out of those constraints when necessary, adding your own custom CSS, and extending the framework with plugins.

Customizing your theme

If you want to change things like your color palette, spacing scale, typography scale, or breakpoints, add your customizations to the `theme` section of your `tailwind.config.js` file:

tailwind.config.js

```
module.exports = {
  theme: {
    screens: {
      sm: '480px',
      md: '768px',
      lg: '976px',
      xl: '1440px',
```

```
    },
    colors: {
      'blue': '#1fb6ff',
      'pink': '#ff49db',
      'orange': '#ff7849',
      'green': '#13ce66',
      'gray-dark': '#273444',
      'gray': '#8492a6',
      'gray-light': '#d3dce6',
    },
    fontFamily: {
      sans: ['Graphik', 'sans-serif'],
      serif: ['Merriweather', 'serif'],
    },
    extend: {
      spacing: {
        '128': '32rem',
        '144': '36rem',
      },
      borderRadius: {
        '4xl': '2rem',
      }
    }
  }
}
```

Learn more about customizing your theme in the **Theme Configuration** documentation.

Using arbitrary values

While you can usually build the bulk of a well-crafted design using a constrained set of design tokens, once in a while you need to break out of those constraints to get things pixel-perfect.

When you find yourself really needing something like `top: 117px` to get a background image in just the right spot, use Tailwind's square bracket notation to generate a class on the fly with any arbitrary value:

```
<div class="top-[117px]">
  <!-- ... -->
</div>
```

This is basically like inline styles, with the major benefit that you can combine it with interactive modifiers like `hover` and responsive modifiers like `lg`:

```
<div class="top-[117px] lg:top-[344px]">
  <!-- ... -->
</div>
```

v3.2.4

Core Concepts › **Adding Custom Styles**

```
  <!-- ... -->
</div>
```

It's even possible to use the **`theme` function** to reference the design tokens in your `tailwind.config.js` file:

```
<div class="grid grid-cols-[fit-content(theme(spacing.32))]">
  <!-- ... -->
</div>
```

Arbitrary properties

If you ever need to use a CSS property that Tailwind doesn't include a utility for out of the box, you can also use square bracket notation to write completely arbitrary CSS:

```
<div class="[mask-type:luminance]">
  <!-- ... -->
</div>
```

This is *really* like inline styles, but again with the benefit that you can use modifiers:

```
<div class="[mask-type:luminance] hover:[mask-type:alpha]">
  <!-- ... -->
</div>
```

This can be useful for things like CSS variables as well, especially when they need to change under different conditions:

```
<div class="[--scroll-offset:56px] lg:[--scroll-offset:44px]">
  <!-- ... -->
</div>
```

Arbitrary variants

Arbitrary *variants* are like arbitrary values but for doing on-the-fly selector modification, like you can with built-in pseudo-class variants like `` `hover:{utility}` `` or responsive variants like `` `md:{utility}` `` but using square bracket notation directly in your HTML.

HTML | Generated CSS

```
<ul role="list">
  {#each items as item}
    <li class="lg:[&:nth-child(3)]:hover:underline">{item}</li>
```

```
  {/each}
</ul>
```

Learn more in the **arbitrary variants** documentation.

Handling whitespace

When an arbitrary value needs to contain a space, use an underscore (`_`) instead and Tailwind will automatically convert it to a space at build-time:

```
<div class="grid grid-cols-[1fr_500px_2fr]">
  <!-- ... -->
</div>
```

In situations where underscores are common but spaces are invalid, Tailwind will preserve the underscore instead of converting it to a space, for example in URLs:

```
<div class="bg-[url('/what_a_rush.png')]">
  <!-- ... -->
</div>
```

In the rare case that you actually need to use an underscore but it's ambiguous because a space is valid as well, escape the underscore with a backslash and Tailwind won't convert it to a space:

```
<div class="before:content-['hello\_world']">
  <!-- ... -->
</div>
```

If you're using something like JSX where the backslash is stripped from the rendered HTML, use **String.raw()** so the backslash isn't treated as a JavaScript escape character:

```
<div className={String.raw`before:content-['hello\_world']`}>
  <!-- ... -->
</div>
```

Resolving ambiguities

Many utilities in Tailwind share a common namespace but map to different CSS properties. For example `text-lg` and `text-black` both share the `text-` namespace, but one is for `font-size` and the other is for `color`.

When using arbitrary values, Tailwind can generally handle this ambiguity automatically based on the value you pass in:

```
<!-- Will generate a font-size utility -->
<div class="text-[22px]">...</div>

<!-- Will generate a color utility -->
<div class="text-[#bada55]">...</div>
```

Sometimes it really is ambiguous though, for example when using CSS variables:

```
<div class="text-[var(--my-var)]">...</div>
```

In these situations, you can "hint" the underlying type to Tailwind by adding a **CSS data type** before the value:

```
<!-- Will generate a font-size utility -->
<div class="text-[length:var(--my-var)]">...</div>

<!-- Will generate a color utility -->
<div class="text-[color:var(--my-var)]">...</div>
```

Using CSS and @layer

When you need to add truly custom CSS rules to a Tailwind project, the easiest approach is to just add the custom CSS to your stylesheet:

main.css

```
@tailwind base;
@tailwind components;
@tailwind utilities;

.my-custom-style {
  /* ... */
}
```

For more power, you can also use the `@layer` directive to add styles to Tailwind's `base`, `components`, and `utilities` layers:

main.css

```
@tailwind base;
@tailwind components;
@tailwind utilities;

@layer components {
  .my-custom-style {
    /* ... */
  }
}
```

▶ Why does Tailwind group styles into "layers"?

The `@layer` directive helps you control declaration order by automatically relocating your styles to the corresponding `@tailwind` directive, and also enables features like **modifiers** and **tree-shaking** for your own custom CSS.

Adding base styles

If you just want to set some defaults for the page (like the text color, background color, or font family), the easiest option is just adding some classes to the `html` or `body` elements:

```
<!doctype html>
<html lang="en" class="text-gray-900 bg-gray-100 font-serif">
  <!-- ... -->
</html>
```

This keeps your base styling decisions in your markup alongside all of your other styles, instead of hiding them in a separate file.

If you want to add your own default base styles for specific HTML elements, use the `@layer` directive to add those styles to Tailwind's `base` layer:

main.css

```
@tailwind base;
@tailwind components;
@tailwind utilities;

@layer base {
  h1 {
    @apply text-2xl;
  }
  h2 {
    @apply text-xl;
  }
  /* ... */
}
```

Use the `theme` function or `@apply` directive when adding custom base styles if you want to refer to any of the values defined in your **theme**.

Adding component classes

Use the `components` layer for any more complicated classes you want to add to your project that you'd still like to be able to override with utility classes.

Traditionally these would be classes like `card`, `btn`, `badge` — that kind of thing.

main.css

```
@tailwind base;
@tailwind components;
@tailwind utilities;

@layer components {
  .card {
    background-color: theme('colors.white');
    border-radius: theme('borderRadius.lg');
    padding: theme('spacing.6');
    box-shadow: theme('boxShadow.xl');
  }
  /* ... */
}
```

By defining component classes in the `components` layer, you can still use utility classes to override them when necessary:

```
<!-- Will look like a card, but with square corners -->
<div class="card rounded-none">
  <!-- ... -->
</div>
```

Using Tailwind you probably don't need these types of classes as often as you think. Read our guide on **Reusing Styles** for our recommendations.

The `components` layer is also a good place to put custom styles for any third-party components you're using:

main.css

```
@tailwind base;
@tailwind components;
@tailwind utilities;

@layer components {
  .select2-dropdown {
    @apply rounded-b-lg shadow-md;
  }
  .select2-search {
    @apply border border-gray-300 rounded;
  }
  .select2-results__group {
    @apply text-lg font-bold text-gray-900;
  }
  /* ... */
}
```

Use the `theme` function or `@apply` directive when adding custom component styles if you want to refer to any of the values defined in your **theme**.

Adding custom utilities

Add any of your own custom utility classes to Tailwind's `utilities` layer:

main.css

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

```
@layer utilities {
  .content-auto {
    content-visibility: auto;
  }
}
```

This can be useful when there's a CSS feature you'd like to use in your project that Tailwind doesn't include utilities for out of the box.

Using modifiers with custom CSS

Any custom styles you add to Tailwind with `@layer` will automatically support Tailwind's modifier syntax for handling things like hover states, responsive breakpoints, dark mode, and more.

CSS

```
@tailwind base;
@tailwind components;
@tailwind utilities;

@layer utilities {
  .content-auto {
    content-visibility: auto;
  }
}
```

HTML

```
<div class="lg:dark:content-auto">
  <!-- ... -->
</div>
```

Learn more about how these modifiers work in the **Hover, Focus, and Other States** documentation.

Removing unused custom CSS

Any custom styles you add to the `base`, `components`, or `utilities` layers will only be included in your compiled CSS if those styles are actually used in your HTML.

main.css

```
@tailwind base;
@tailwind components;
@tailwind utilities;

@layer components {
  /* This won't be included in your compiled CSS unless you actually use it */
  .card {
    /* ... */
  }
}
```

If you want to add some custom CSS that should always be included, add it to your stylesheet without using the `@layer` directive:

main.css

```
@tailwind base;
@tailwind components;

/* This will always be included in your compiled CSS */
.card {
  /* ... */
}

@tailwind utilities;
```

Make sure to put your custom styles where they need to go to get the precedence behavior you want. In the example above, we've added the `.card` class before `@tailwind utilities` to make sure utilities can still override it.

Using multiple CSS files

If you are writing a lot of CSS and organizing it into multiple files, make sure those files are combined into a single stylesheet before processing them with Tailwind, or you'll see errors about using `@layer` without the corresponding `@tailwind` directive.

The easiest way to do this is using the **postcss-import** plugin:

postcss.config.js

```
  module.exports = {
    plugins: {
+     'postcss-import': {},
      tailwindcss: {},
      autoprefixer: {},
    }
  }
```

Learn more in our **build-time imports** documentation.

Layers and per-component CSS

Component frameworks like Vue and Svelte support adding per-component styles within a `<style>` block that lives in each component file.

While you can use features like `@apply` and `theme` inside component styles like this, the `@layer` directive will not work and you'll see an error about `@layer` being used without a matching `@tailwind` directive:

Don't use `@layer` in component styles

Card.svelte

```
<div>
  <slot></slot>
</div>

<style>
  /* Won't work because this file is processed in isolation */
  @layer components {
    div {
      background-color: theme('colors.white');
      border-radius: theme('borderRadius.lg');
      padding: theme('spacing.6');
      box-shadow: theme('boxShadow.xl');
    }
  }
</style>
```

This is because under-the-hood, frameworks like Vue and Svelte are processing every single `<style>` block independently, and running your PostCSS plugin chain against each one in isolation.

That means if you have 10 components that each have a `<style>` block, Tailwind is being run 10 separate times, and each run has zero knowledge about the other runs. Because of this, Tailwind can't take the styles you define in a `@layer` and move them to the

corresponding `@tailwind` directive, because as far as Tailwind can tell there is no `@tailwind` directive to move it to.

One solution to this is to simply *not* use `@layer` inside your component styles:

Add your styles without using `@layer`

Card.svelte

```
<div>
  <slot></slot>
</div>

<style>
  div {
    background-color: theme('colors.white');
    border-radius: theme('borderRadius.lg');
    padding: theme('spacing.6');
    box-shadow: theme('boxShadow.xl');
  }
</style>
```

You lose the ability to control the precedence of your styles, but unfortunately that's totally out of our control because of how these tools work.

Our recommendation is that you just don't use component styles like this at all and instead use Tailwind the way it's intended to be used — as a single global stylesheet where you use the classes directly in your HTML:

Use Tailwind's utilities instead of component styles

Card.svelte

```
<div class="bg-white rounded-lg p-6 shadow-xl">
  <slot></slot>
</div>
```

Writing plugins

You can also add custom styles to your project using Tailwind's plugin system instead of using a CSS file:

tailwind.config.js

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

module.exports = {
  // ...
  plugins: [
    plugin(function ({ addBase, addComponents, addUtilities, theme }) {
      addBase({
        'h1': {
          fontSize: theme('fontSize.2xl'),
        },
        'h2': {
          fontSize: theme('fontSize.xl'),
        },
      })
      addComponents({
        '.card': {
          backgroundColor: theme('colors.white'),
          borderRadius: theme('borderRadius.lg'),
          padding: theme('spacing.6'),
          boxShadow: theme('boxShadow.xl'),
        }
      })
      addUtilities({
        '.content-auto': {
          contentVisibility: 'auto',
        }
      })
    })
  ]
}
```

Learn more about writing your own plugins in the **Plugins** documentation.

‹ Reusing Styles

Functions & Directives ›

Edit this page on GitHub