# Get started with Tailwind CSS

Tailwind CSS works by scanning all of your HTML files, JavaScript components, and any other templates for class names, generating the corresponding styles and then writing them to a static CSS file.

It's fast, flexible, and reliable — with zero-runtime.

## Installation

The simplest and fastest way to get up and running with Tailwind CSS from scratch is with the Tailwind CLI tool. The CLI is also available as a standalone executable if you want to use it without installing Node.js.

1. Install Tailwind CSS

Install tailwindcss via npm, and create your tailwind.config.js file.

Terminal:

npm install -D tailwindcss

npx tailwindcss init

1. Configure your template paths

Add the paths to all of your template files in your tailwind.config.js file.

Tailwind.config.js

/\*\* @type {import('tailwindcss').Config} \*/

module.exports = {

content: ["./src/\*\*/\*.{html,js}"],

theme: {

extend: {},

},

plugins: [],

}

1. Add the Tailwind directives to your CSS

Add the @tailwind directives for each of Tailwind’s layers to your main CSS file.

Src/input.css:

@tailwind base;

@tailwind components;

@tailwind utilities;

1. Start the Tailwind CLI build process

Run the CLI tool to scan your template files for classes and build your CSS.

Terminal:

npx tailwindcss -i ./src/input.css -o ./dist/output.css –watch

1. Start using Tailwind in your HTML

Add your compiled CSS file to the <head> and start using Tailwind’s utility classes to style your content.

Src/index.html

<!doctype html>

<html>

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<link href="/dist/output.css" rel="stylesheet">

</head>

<body>

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

Hello world!

</h1>

</body>

</html>

# Utility-First Fundamentals

Building complex components from a constrained set of primitive utilities.

Traditionally, whenever you need to style something on the web, you write CSS.

With Tailwind, you style elements by applying pre-existing classes directly in your HTML.

This approach allows us to implement a completely custom component design without writing a single line of custom CSS.

Now I know what you’re thinking, “this is an atrocity, what a horrible mess!” and you’re right, it’s kind of ugly. In fact it’s just about impossible to think this is a good idea the first time you see it — you have to actually try it.

But once you’ve actually built something this way, you’ll quickly notice some really important benefits:

* You aren’t wasting energy inventing class names. No more adding silly class names like sidebar-inner-wrapper just to be able to style something, and no more agonizing over the perfect abstract name for something that’s really just a flex container.
* Your CSS stops growing. Using a traditional approach, your CSS files get bigger every time you add a new feature. With utilities, everything is reusable so you rarely need to write new CSS.
* Making changes feels safer. CSS is global and you never know what you’re breaking when you make a change. Classes in your HTML are local, so you can change them without worrying about something else breaking.

When you realize how productive you can be working exclusively in HTML with predefined utility classes, working any other way will feel like torture.

# Responsive Design

Using responsive utility variants to build adaptive user interfaces.

Every utility class in Tailwind can be applied conditionally at different breakpoints, which makes it a piece of cake to build complex responsive interfaces without ever leaving your HTML.

There are five breakpoints by default, inspired by common device resolutions:

A screenshot of a computer

Description automatically generated with medium confidence

To add a utility but only have it take effect at a certain breakpoint, all you need to do is prefix the utility with the breakpoint name, followed by the : character:

<!-- Width of 16 by default, 32 on medium screens, and 48 on large screens -->

<img class="w-16 md:w-32 lg:w-48" src="...">

## Working mobile-first

By default, Tailwind uses a mobile-first breakpoint system, similar to what you might be used to in other frameworks like Bootstrap.

What this means is that unprefixed utilities (like uppercase) take effect on all screen sizes, while prefixed utilities (like md:uppercase) only take effect at the specified breakpoint and above.

## Targeting a breakpoint range

By default, styles applied by rules like md:flex will apply at that breakpoint and stay applied at larger breakpoints.

If you’d like to apply a utility only when a specific breakpoint range is active, stack a responsive modifier like md with a max-\* modifier to limit that style to a specific range:

<div class="md:max-xl:flex">

<!-- ... -->

</div>

Tailwind generates a corresponding max-\* modifier for each breakpoint, so out of the box the following modifiers are available:

A screenshot of a computer program

Description automatically generated with low confidence

## Targeting a single breakpoint

To target a single breakpoint, target the range for that breakpoint by stacking a responsive modifier like md with the max-\* modifier for the next breakpoint:

<div class="md:max-lg:flex">

<!-- ... -->

</div>

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',

}

}

}

}

# Dark Mode

Now that dark mode is a first-class feature of many operating systems, it’s becoming more and more common to design a dark version of your website to go along with the default design.

To make this as easy as possible, Tailwind includes a dark variant that lets you style your site differently when dark mode is enabled:

A screenshot of a phone

Description automatically generated with medium confidence

<div class="bg-white dark:bg-slate-800 rounded-lg px-6 py-8 ring-1 ring-slate-900/5 shadow-xl">

<div>

<span class="inline-flex items-center justify-center p-2 bg-indigo-500 rounded-md shadow-lg">

<svg class="h-6 w-6 text-white" xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 24 24" stroke="currentColor" aria-hidden="true"><!-- ... --></svg>

</span>

</div>

<h3 class="text-slate-900 dark:text-white mt-5 text-base font-medium tracking-tight">Writes Upside-Down</h3>

<p class="text-slate-500 dark:text-slate-400 mt-2 text-sm">

The Zero Gravity Pen can be used to write in any orientation, including upside-down. It even works in outer space.

</p>

</div>

## Toggling dark mode manually

If you want to support toggling dark mode manually instead of relying on the operating system preference, use the class strategy instead of the media strategy:

Tailwind.config.js

module.exports = {

darkMode: 'class',

// ...

}

Now instead of dark:{class} classes being applied based on prefers-color-scheme, they will be applied whenever dark class is present earlier in the HTML tree.

<!-- Dark mode not enabled -->

<html>

<body>

<!-- Will be white -->

<div class="bg-white dark:bg-black">

<!-- ... -->

</div>

</body>

</html>

<!-- Dark mode enabled -->

<html class="dark">

<body>

<!-- Will be black -->

<div class="bg-white dark:bg-black">

<!-- ... -->

</div>

</body>

</html>

# Functions & Directives

A reference for the custom functions and directives Tailwind exposes to your CSS.

​

## Directives

Directives are custom Tailwind-specific at-rules you can use in your CSS that offer special functionality for Tailwind CSS projects.

​

## @tailwind

Use the @tailwind directive to insert Tailwind’s base, components, utilities and variants styles into your CSS.

/\*\*

\* This injects Tailwind's base styles and any base styles registered by

\* plugins.

\*/

@tailwind base;

/\*\*

\* This injects Tailwind's component classes and any component classes

\* registered by plugins.

\*/

@tailwind components;

/\*\*

\* This injects Tailwind's utility classes and any utility classes registered

\* by plugins.

\*/

@tailwind utilities;

/\*\*

\* Use this directive to control where Tailwind injects the hover, focus,

\* responsive, dark mode, and other variants of each class.

\*

\* If omitted, Tailwind will append these classes to the very end of

\* your stylesheet by default.

\*/

@tailwind variants;

## @layer

Use the @layer directive to tell Tailwind which “bucket” a set of custom styles belong to. Valid layers are base, components, and utilities.

@tailwind base;

@tailwind components;

@tailwind utilities;

@layer base {

h1 {

@apply text-2xl;

}

h2 {

@apply text-xl;

}

}

@layer components {

.btn-blue {

@apply bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4 rounded;

}

}

@layer utilities {

.filter-none {

filter: none;

}

.filter-grayscale {

filter: grayscale(100%);

}

}

Wrapping any custom CSS with @layer also makes it possible to use modifiers with those rules, like hover: and focus: or responsive modifiers like md: and lg:.

## @apply

Use @apply to inline any existing utility classes into your own custom CSS.

This is useful when you need to write custom CSS (like to override the styles in a third-party library) but still want to work with your design tokens and use the same syntax you’re used to using in your HTML.

.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;

}