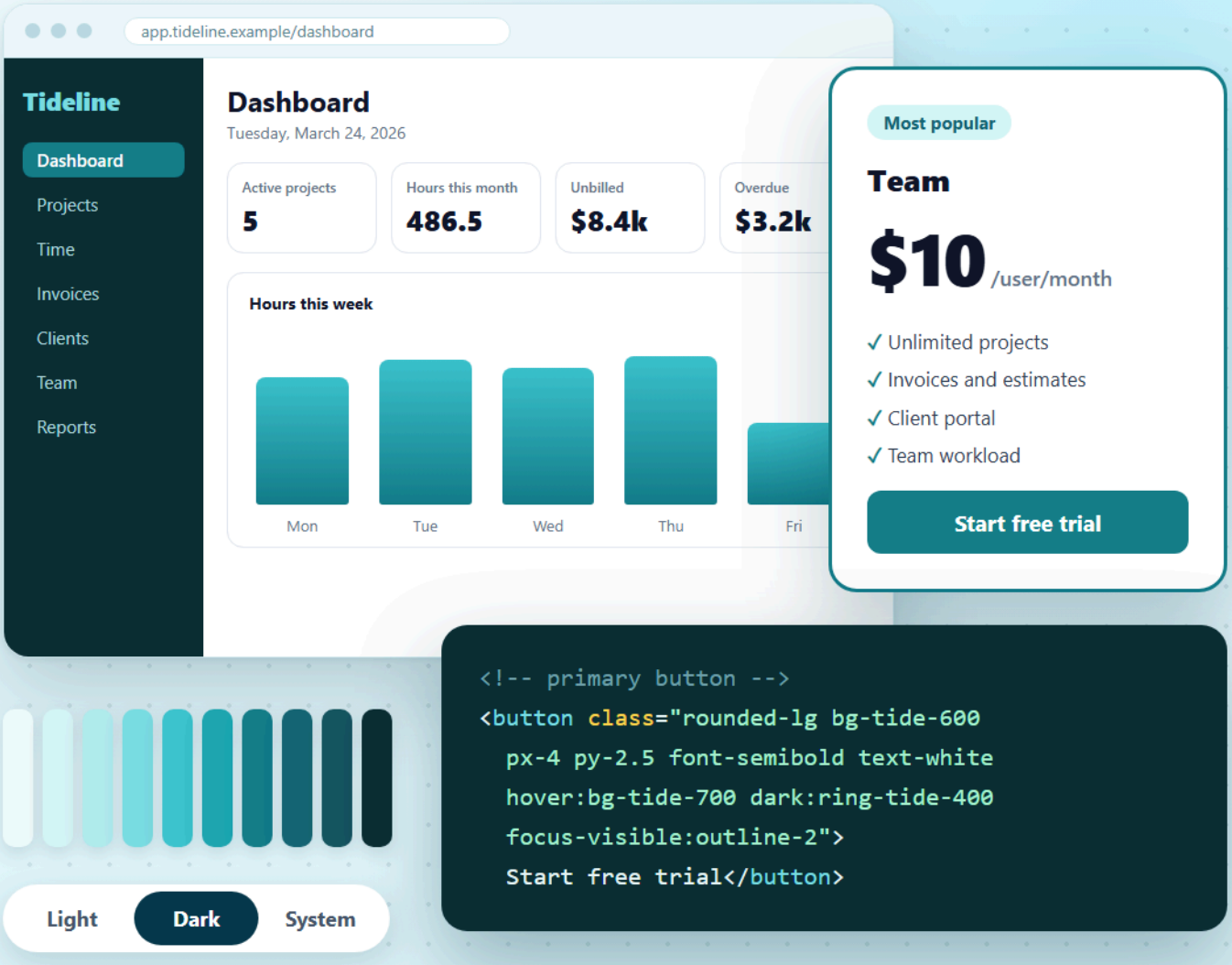


A PRACTICAL, PROJECT-BASED GUIDE

Tailwind CSS: Modern UI Design

Utility-first styling, responsive layouts, components, dark mode, and design systems

Tailwind CSS v4 · @theme tokens · Flexbox & Grid · Forms
Pricing pages · Dashboards · Accessibility · React & Vue

LANDING PAGE · PRICING · AUTH · APP DASHBOARD

THENEXBOOKS

Tailwind CSS: Modern UI Design

Utility-First Styling, Responsive Layouts, Components, Dark Mode, and Design Systems
From your first utility class to a complete SaaS marketing site and app dashboard — built step by step with Tailwind CSS v4

First Edition

THENEXBOOKS

Tailwind CSS: Modern UI Design
Utility-First Styling, Responsive Layouts, Components, Dark Mode, and Design Systems

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples — HTML, CSS, and JavaScript — may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Tailwind CSS, browsers, and the surrounding tools change over time; utility names, defaults, and feature availability can differ between releases. Always consult the official documentation for the version you use.

Tailwind CSS is a trademark of Tailwind Labs Inc. React is a trademark of Meta Platforms, Inc. Vue.js is a trademark of its respective owner. Microsoft Edge, Visual Studio Code, and Windows are trademarks of the Microsoft group of companies. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by Tailwind Labs Inc. The cover and all interface designs in this book are original works and do not reproduce any trademarked logo or commercial component library.

Tideline, the software product used as the running example in this book, is fictional, as are its company, customers, clients, team members, projects, invoices, prices, and all sample data. Any resemblance to real products, companies, or people is coincidental. Email addresses and domains use reserved example domains.

Preface

Most CSS projects start clean and end tangled. The first stylesheet is tidy; a year later it is thousands of lines long, half of it is afraid to be deleted, class names like `.card-new-2` appear, and changing the padding of one button breaks a page nobody remembered. Designers and developers argue about naming more than about design, and every new screen means another round of inventing names and hunting for the right file.

Tailwind CSS takes a different path. Instead of writing new CSS for every component, you compose interfaces from small, single-purpose utility classes — `flex`, `gap-4`, `rounded-lg`, `bg-tide-600`, `hover:bg-tide-700` — that come from one consistent design system. Responsive layouts, hover and focus states, and dark mode are written right where the markup is, and the build step generates only the CSS your project actually uses. The result is faster work, fewer surprises, and interfaces that stay consistent as they grow.

This book teaches Tailwind CSS v4 from the very first class to a complete, production-ready product interface. We do not just list utilities. We build one realistic product — a marketing site and an application dashboard — introducing each idea at the moment it solves a real design problem, and explaining the CSS behind every utility so you always understand what the browser receives.

Who This Book Is For

- **Front-end and full-stack developers** who know HTML and CSS and want to build polished, responsive interfaces much faster.
- **Designers who code** and want a direct, reliable path from a design system to working screens.
- **Developers moving from Bootstrap or hand-written CSS** who want to understand the utility-first approach properly, including when not to use it.
- **Teams already on Tailwind CSS v3** who need to understand the CSS-first configuration of v4 and migrate with confidence.

What You Will Learn

- How utility-first CSS works, how to set up Tailwind CSS v4 with the CLI, Vite, or PostCSS, and how the build finds your classes.
- The core utilities for spacing, sizing, color, typography, borders, shadows, and backgrounds — and the CSS each one produces.
- Variants for every state: hover, focus, groups and peers, ARIA and data attributes, responsive breakpoints, container queries, and dark mode.
- Design tokens with `@theme`: building a brand color scale, fonts, shadows, breakpoints, and animations in pure CSS.

- Layout with flexbox and grid, mobile-first responsive design, and complete page sections such as heroes, feature grids, pricing tables, and footers.
- Components: buttons, badges, forms with validation states, cards, navigation, tables, modals, dropdowns, popovers, and toasts.
- Theming and design systems: dark mode strategies, semantic tokens, custom utilities and variants, component recipes, and a living style guide.
- Production skills: accessibility, motion, framework components in React and Vue, class organization, builds and browser support, and migrating from v3 to v4.

How This Book Is Organized

Part I — Foundations (Chapters 1–5) explains the utility-first idea, sets up Tailwind CSS v4, and covers core utilities, variants, and design tokens with `@theme`. **Part II — Layout and Responsive Design** (Chapters 6–10) covers the box model, flexbox, grid, breakpoints and container queries, and assembles the complete landing page. **Part III — Components** (Chapters 11–15) builds buttons, badges, forms and authentication pages, cards and pricing tables, navigation, and data tables. **Part IV — Theming and Design Systems** (Chapters 16–19) adds dark mode, custom CSS with `@utility` and `@custom-variant`, typography and plugins, and a documented design system. **Part V — The Application UI** (Chapters 20–24) builds the dashboard shell, overlays, motion, accessibility, and framework components. **Part VI — Production** (Chapters 25–28) covers maintainable markup, builds and performance, upgrading from v3, and a complete case study. The appendices provide a cheat sheet, templates and reference tables, a production checklist, and a glossary.

Each chapter ends with exercises and a **Key Takeaways** box for quick review.

The Running Example: Tideline

Throughout the book we design and build the interface for **Tideline**, a fictional software product that helps small creative agencies plan projects, track time, and send invoices. Tideline needs a marketing site that converts visitors — a landing page, a pricing page with monthly and annual plans, and sign-up and log-in pages — and an application that customers use every day: a dashboard with a responsive sidebar, key numbers, a weekly hours chart, project and invoice tables, forms, modal dialogs, and notifications. Inside the app we follow one customer workspace, Brightside Studio, with eight team members and a handful of real-looking projects and invoices. Every screen uses the same brand tokens, copy, and sample data, so the pieces fit together into one coherent product by the final chapter.

Versions and Conventions

Examples target **Tailwind CSS v4** and were checked with the v4.3 command-line tool. Version 4 is configured in CSS with `@import "tailwindcss"` and `@theme`; the JavaScript configuration file of earlier versions is treated as legacy and covered in Chapter 27. Where a feature arrived in a later v4 release or

depends on newer browser support, a Note says so. Tailwind CSS v4 targets current browsers (Safari 16.4+, Chrome 111+, Firefox 128+). Utility names and defaults can change between releases; when something does not behave as described, check the release notes for your version.

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version differences, or a useful aside.

> **WARNING**
> A common mistake or something that can cause broken layouts, inaccessible interfaces, or hard-to-find bugs.

Code appears in `monospace`; long class lists are wrapped over several lines to fit the page, which is also how we recommend writing them in real projects. File captions above code blocks, such as `tideline-ui/pricing.html`, show where the code belongs. Generated CSS is shown as an excerpt and may be formatted slightly differently from your build output.

Contents

Preface

PART I — FOUNDATIONS

1 Why Utility-First? How Tailwind CSS Thinks

2 Setting Up Tailwind CSS v4: The CLI, Vite, and Your First Build

3 Core Utilities: Spacing, Sizing, Color, and Typography

4 Variants and States: Hover, Focus, Responsive, and Beyond

5 Design Tokens with @theme: Building the Tideline Brand

PART II — LAYOUT AND RESPONSIVE DESIGN

6 The Box Model in Utilities: Borders, Radius, Shadows, and Backgrounds

7 Flexbox Layouts: Alignment, Spacing, and Real Patterns

8 Grid Layouts: Columns, Spans, and Page Structure

9 Responsive Design: Mobile-First Breakpoints and Container Queries

10 Building the Landing Page: Hero, Features, Testimonials, and Footer

PART III — COMPONENTS

11 Buttons, Links, and Badges: Your First Reusable Patterns

12 Forms and Auth Pages: Inputs, Validation States, and Accessible Labels

13 Cards, Pricing Tables, and Feature Grids

14 Navigation: Navbars, Mobile Menus, Tabs, and Breadcrumbs

15 Tables, Lists, and Data Display

PART IV — THEMING AND DESIGN SYSTEMS

16 Dark Mode: Strategies, Variants, and a Theme Switcher

17 Extending Tailwind in CSS: @utility, @custom-variant, @layer, and @apply

18 Typography, Rich Content, and Plugins

19 Building a Design System: Semantic Tokens, Component Recipes, and Documentation

PART V — THE APPLICATION UI

20 The Dashboard Shell: Sidebar, Top Bar, and Responsive App Layout

21 Overlays: Modals, Dropdowns, Popovers, and Toasts

22 Motion: Transitions, Animations, and Reduced Motion

23 Accessible Interfaces: Focus, Contrast, ARIA Variants, and Screen Readers

24 Tailwind with Components: React, Vue, and Class Variants

PART VI — PRODUCTION

25 Clean, Maintainable Markup: Class Order, Prettier, and Avoiding Class Soup

26 Builds and Performance: Source Detection, Minification, and Browser Support

27 Upgrading and Migrating: From v3 to v4 and Beyond

28 Case Study: Shipping the Complete Tideline Site and Dashboard

APPENDICES

A Tailwind CSS Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

Foundations

How utility-first CSS works and why it scales, setting up Tailwind CSS v4 and your first build, the core utilities for spacing, sizing, color, and type, variants for every state, and the design tokens that give Tideline its brand.

1 · Why Utility-First?
2 · Setting Up Tailwind CSS v4
3 · Core Utilities
4 · Variants and States
5 · Design Tokens with @theme

CHAPTER 1 · PART I — FOUNDATIONS

Why Utility-First? How Tailwind CSS Thinks

Before we write a single build command, we need to understand the idea that makes Tailwind CSS different: styling with small, single-purpose classes chosen from a design system. In this chapter we look at why traditional stylesheets get harder to maintain as a project grows, rebuild one Tideline project card both ways, answer the most common objections honestly, and map out the product we will build together through the rest of the book.

The Problem With Traditional CSS at Scale

Almost every web developer has lived through the same story. A project starts with a clean stylesheet. There is a `.header`, a `.button`, a `.card`, and everything feels tidy. Six months later the file has grown to thousands of lines, nobody is quite sure which rules are still used, and every new feature starts with the same nervous question: *if I change this, what else breaks?*

The problem is not that the developers were careless. It is that the traditional way of writing CSS — invent a meaningful class name, then describe its appearance in a separate file — has costs that grow with the size of the project and the size of the team. Let us name those costs precisely, because utility-first CSS is a direct answer to each of them.

Naming is hard, and it never stops

Every new piece of interface needs a name. Is the small gray text under a project title a `.card-subtitle`, a `.project-meta`, or a `.text-muted`? Methodologies such as BEM (Block, Element, Modifier) help by giving names a predictable shape — `.project-card__meta` — but they do not remove the decision. Each name is a small design document, and each one has to be invented, remembered, and kept consistent across the codebase. On a team, two people will name the same thing differently, and you end up with `.card-subtitle` and `.card__sub` doing nearly identical work.

Specificity and the cascade fight back

CSS rules compete. A selector with an ID beats one with classes, two classes beat one, and later rules beat earlier ones with equal weight. In a small file you can keep that in your head. In a large one, you add a rule, it does not apply, and you reach for a longer selector or `!important` to win the fight. The next person does the same, and the stylesheet slowly turns into an arms race where the most specific rule wins rather than the most sensible one.

Dead CSS and the append-only stylesheet

Deleting CSS is frightening. A class name might be used in a template you have not opened, built dynamically in JavaScript, or referenced by a page that only appears once a year. Because removing a rule is risky and adding one is safe, teams add. Stylesheets become append-only: they grow with every feature and rarely shrink, even when the features they styled are long gone.

Context switching

To build one component in the traditional way, you jump between at least two files: the markup, where you invent a class name, and the stylesheet, where you scroll to find (or create) the matching rule. When you later read the markup, the class name tells you *what* the element is but not *how it looks*; you must open the stylesheet again to find out. That back-and-forth is small each time, but it adds up over hundreds of components.

Inconsistent values

Finally, a hand-written stylesheet accepts any value. One developer writes `padding: 14px`, another writes `padding: 0.9rem`, a third writes `padding: 15px`. Each looks fine alone, but together they create an interface that feels subtly uneven. Design systems exist to fix exactly this, but a plain stylesheet has no way to enforce one.

What Utility-First Means

Utility-first CSS turns the usual workflow around. Instead of inventing a class name for each component and then writing CSS for it, you compose the design directly in your markup from a large set of small, predefined classes. Each class does one job. `p-6` sets padding. `text-sm` sets a font size and a matching line height. `rounded-2xl` rounds the corners. `bg-white` sets a background color.

The important word is not "small" but **predefined**. The classes are generated from a design system: a spacing scale, a type scale, a color palette, a set of radii and shadows. When you write `p-6`, you are not typing an arbitrary number; you are choosing step 6 on a spacing scale that every other element also uses. That is what makes utility-first different from writing inline styles, as we will see shortly.

It helps to see the actual CSS that Tailwind CSS produces. Here are a few utilities from the card we are about to build, exactly as the v4.3 command-line tool generated them:

Generated CSS (excerpt, Tailwind CSS v4.3)

```
.p-6 {
  padding: calc(var(--spacing) * 6);
}
.text-sm {
  font-size: var(--text-sm);
  line-height: var(--tw-leading, var(--text-sm--line-height));
}
.rounded-2xl {
  border-radius: var(--radius-2xl);
```

```
}
.bg-white {
  background-color: var(--color-white);
}
```

Notice that nothing is hard-coded. Padding is a multiple of one variable, `--spacing`, which defaults to `0.25rem`, so `p-6` works out to `1.5rem`. The font size, radius, and color all come from CSS custom properties that Tailwind defines once, in a theme layer, at the top of the generated file. Change the variable and every utility that uses it follows. In Chapter 5 we will define our own variables — Tideline's brand colors, font, and card shadow — and Tailwind will create matching utilities for them automatically.

> **NOTE**
> "Utility class" and "atomic class" mean roughly the same thing: a class that applies one small, reusable piece of styling. Tailwind CSS calls them utilities, and calls the prefixes that make them conditional (`hover:`, `md:`, `dark:`) **variants**. We use those two words throughout the book.

One Tideline Card, Built Two Ways

Let us make this concrete. Throughout the book we build Tideline, a fictional product for small creative agencies. One of the first things a Tideline user sees in the app is a list of project cards. Here is the card for project PRJ-1042, a website redesign for the client Harborline Coffee, due on April 18, 2026, and 64% complete.

The traditional way: semantic class names

First, the approach most of us learned. We give each part of the card a descriptive BEM-style name and write the styles in a stylesheet.

`project-card.html (traditional)`

```
<article class="project-card">
  <div class="project-card__header">
    <span class="project-card__id">PRJ-1042</span>
    <span class="project-card__badge">In progress</span>
  </div>
  <h3 class="project-card__title">Website Redesign</h3>
  <p class="project-card__meta">Harborline Coffee · Due Apr 18, 2026</p>
  <div class="project-card__bar">
    <div class="project-card__fill" style="width: 64%"></div>
  </div>
</article>
```

project-card.css (traditional)

```
.project-card {
  max-width: 24rem;
  padding: 1.5rem;
  border-radius: 1rem;
  background: #fff;
  box-shadow: 0 1px 2px rgb(15 23 42 / 0.06), 0 0 0 1px #e2e8f0;
}
.project-card__header {
  display: flex;
  align-items: center;
  justify-content: space-between;
  gap: 1rem;
}
.project-card__id {
  font-family: ui-monospace, monospace;
  font-size: 0.75rem;
  color: #64748b;
}
.project-card__badge {
  display: inline-flex;
  padding: 0.125rem 0.5rem;
  border-radius: 9999px;
  font-size: 0.75rem;
  font-weight: 500;
  background: #f0fdfa;
  color: #0f766e;
}
.project-card__title {
  margin-top: 0.75rem;
  font-size: 1.125rem;
  font-weight: 600;
  color: #0f172a;
}
.project-card__meta {
  margin-top: 0.25rem;
  font-size: 0.875rem;
  color: #475569;
}
.project-card__bar {
  margin-top: 1rem;
  height: 0.5rem;
  border-radius: 9999px;
  background: #f1f5f9;
}
.project-card__fill {
  height: 100%;
  border-radius: 9999px;
  background: #14b8a6;
}
```

There is nothing wrong with this code. It is readable, the names are sensible, and the markup is short. But look at what it required: eight invented class names, a second file, and a long list of raw values — `0.125rem`, `#64748b`, `9999px` — that nothing checks against a shared scale. The next card someone builds will need another set of names and will probably pick slightly different values.

The utility-first way

Now the same card with Tailwind CSS utilities. There is no separate stylesheet to write; the generated CSS file contains everything.

project-card.html (utility-first)

```
<article class="max-w-sm rounded-2xl bg-white p-6 shadow-xs ring-1 ring-slate-200">
  <div class="flex items-center justify-between gap-4">
    <span class="font-mono text-xs text-slate-500">PRJ-1042</span>
    <span class="inline-flex rounded-full bg-teal-50 px-2 py-0.5 text-xs font-medium
                 text-teal-700">In progress</span>
  </div>
  <h3 class="mt-3 text-lg font-semibold text-slate-900">Website Redesign</h3>
  <p class="mt-1 text-sm text-slate-600">Harborline Coffee · Due Apr 18, 2026</p>
  <div class="mt-4 h-2 rounded-full bg-slate-100">
    <div class="h-full w-[64%] rounded-full bg-teal-500"></div>
  </div>
</article>
```

When we built both versions side by side with the v4.3 CLI and took a screenshot, they were practically indistinguishable: a white card with soft rounded corners and a thin light-gray outline, a small monospaced "PRJ-1042" at the top left, a pale teal "In progress" pill at the top right, a bold dark "Website Redesign" title, a gray line of client and due-date text, and a teal progress bar filled to roughly two thirds of its track. The utility version contains no invented names and no second file, and every value except the progress width comes from a shared scale.

Here is how to read the classes on the outer `<article>`. You will learn every one of these in detail in Chapters 3 and 6, but most of them are guessable already:

Utility	CSS it produces	What it does here
`max-w-sm`	`max-width: var(--container-sm)` (24rem)	Keeps the card narrow
`rounded-2xl`	`border-radius: var(--radius-2xl)` (1rem)	Soft corners
`bg-white`	`background-color: var(--color-white)`	White surface
`p-6`	`padding: calc(var(--spacing) * 6)` (1.5rem)	Inner spacing
`shadow-xs`	A very small `box-shadow`	Subtle lift
`ring-1 ring-slate-200`	A 1px `box-shadow` outline in slate-200	Thin border that does not affect layout

The one odd-looking class is `w-[64%]`. The square brackets mark an **arbitrary value**: a one-off value that is not on any scale. A progress percentage is genuinely data, not design, so an arbitrary value (or an inline `style` attribute) is the right tool. We discuss when arbitrary values are appropriate, and when they are a warning sign, in Chapter 3.

> **NOTE**
>
> In Part I we use Tailwind's built-in `teal` palette as a stand-in brand color. In Chapter 5 we define Tideline's own `tide` color scale with `@theme`, and from then on you will see classes such as `bg-tide-600` instead of `bg-teal-600`. The idea is identical; only the color name changes.

What Changes When You Work This Way

Comparing the two versions line by line misses the bigger shift. The real differences show up over months of work on a growing product.

- **You stop naming things that do not need names.** The card still has meaning — it is an `<article>` with a heading — but the wrapper `<div>` that lays out the ID and the badge does not need a name like `.project-card__header`. Names are saved for the things that deserve them: components in your templates or framework, which we discuss below.
- **Your CSS stops growing.** Once `p-6` exists in the generated file, using it on a hundred more elements adds nothing to the stylesheet. The size of the CSS depends on how many *different* utilities you use, not on how many pages you build.
- **Deleting is safe.** When you delete a component's markup, its styles go with it. There is no orphaned rule left behind in a stylesheet, because the next build simply does not generate utilities that are no longer used.
- **Changes are local.** Changing `p-6` to `p-8` on one card affects that card and nothing else. There is no cascade fight to win, because utilities are flat, single-class selectors with the same specificity.
- **Consistency becomes the default.** It is easier to type `p-6` than to invent a new value, so people naturally stay on the scale. The design system is enforced by convenience rather than by code review.

> **TIP**
>
> When you read utility-heavy markup for the first time, read the class list in groups rather than word by word: layout (`flex items-center justify-between`), box (`rounded-2xl p-6 ring-1`), color (`bg-white text-slate-900`), and type (`text-sm font-semibold`). After a few days most developers find they can "see" the element from its classes without opening a browser. Chapter 25 shows how a formatter can sort classes into a consistent order to make this even easier.

The Common Objections, Answered Honestly

Utility-first CSS provokes strong reactions, and some of the criticism is fair. Let us look at the objections you are most likely to hear — or to feel yourself right now — and give honest answers rather than marketing ones.

"The HTML is ugly"

It is true that a utility-styled element carries a long `class` attribute, and the first time you see one it can look like noise. There are three honest responses. First, the information in that attribute has to live somewhere; with traditional CSS it lives in a stylesheet you cannot see while reading the markup. Second, the noise fades with familiarity, the same way a new programming language's syntax stops looking strange after a week. Third, and most practically, you rarely repeat long class lists by hand in a real project. You put the markup in a component or template partial once and reuse that. Long class strings are easier to live with when each one appears in exactly one place.

"Isn't this just inline styles?"

This is the most common objection, and the answer is a clear no. Inline styles and utilities look similar — both put styling in the markup — but they behave very differently:

Capability	Inline `style="..."`	Tailwind utilities
Constrained to a design system	No — any value goes	Yes — scales for spacing, color, type, radius
Hover, focus, and other states	Not possible	Yes — `hover:bg-teal-700`
Responsive breakpoints	Not possible	Yes — `md:flex`
Dark mode, print, reduced motion	Not possible	Yes — `dark:`, `print:`, `motion-reduce:`
Pseudo-elements like `::placeholder`	Not possible	Yes — `placeholder:text-slate-400`
Specificity	Very high; hard to override	One class, easy to override

An inline style cannot say "a darker background when hovered" or "two columns on wide screens", because those require selectors and media queries, and the `style` attribute has neither. A utility like `md:hover:bg-teal-700` compiles to a real rule inside a media query with a `:hover` selector. The constraint aspect matters just as much: inline styles let every developer invent values; utilities steer everyone toward the same ones.

"How do I reuse styles?"

In traditional CSS, the class name is the unit of reuse: you write `.btn` once and use it everywhere. With utilities, the unit of reuse moves up a level to the **component** or **template**. If your site is built with a framework such as React or Vue, a `Button` component holds the classes once. If it is built with a server-side template language, a partial or macro does the same job. If you only have static HTML, your editor's multi-cursor editing handles small repetitions, and loops in whatever generates your pages handle large ones.

For a small, stable set of patterns — buttons, form inputs, badges, cards — Tailwind also lets you define traditional-looking classes built from utilities, using `@layer components` and `@apply`. We build exactly that set for Tideline in Chapter 17 and finish it in Chapter 19. The book's stance is simple: prefer components and partials for reuse, and use CSS component classes only for the handful of patterns that truly appear everywhere. We will explain the trade-off when we get there.

"What about separation of concerns?"

The traditional argument says HTML should describe content and CSS should describe presentation, and mixing them is a mistake. In practice, though, a class like `.project-card__header` is already coupled tightly to the markup: it exists only because that specific `<div>` exists, and changing the structure of the card means changing both files. Utility-first CSS accepts that coupling openly and puts the styling where the structure is. The concern that is genuinely separated is the *design system* — the scales and tokens — which lives in one CSS file and drives every utility.

> **WARNING**
>
> Utilities do not replace knowing CSS. Tailwind CSS is a fast way to *write* CSS, not a way to avoid understanding it. If you do not know what `min-width: 0` does to a flex item, `min-w-0` will not help you. Throughout this book we show the CSS behind each utility precisely so that your CSS knowledge keeps growing as you use it.

When Tailwind CSS Is Not a Good Fit

No tool is right for every project, and you will make better decisions if you know where utility-first CSS struggles. Here are the situations where we would think twice.

- **Content you do not control.** Blog posts written in Markdown or a CMS rich-text editor produce plain `<h2>`, `<p>`, and `<ul>` elements with no classes, and you cannot add utilities to them. Tailwind handles this with a typography plugin or a few scoped rules — we cover both in Chapter 18 — but if a site is almost entirely CMS content, the utility-first benefits shrink.
- **Older browsers.** Tailwind CSS v4 relies on modern CSS features such as cascade layers, `@property`, and `color-mix()`. It targets Safari 16.4+, Chrome 111+, and Firefox 128+. If you must support browsers older than that, stay on Tailwind CSS v3.4, which we discuss in Chapter 27.

- **HTML email.** Email clients have very limited and inconsistent CSS support, and email layouts usually need styles inlined onto each element. Tailwind can be part of an email toolchain, but it is not a drop-in fit, and modern v4 output is not designed with email clients in mind.
- **A team committed to another approach.** If your team already has a mature, well-documented component library with its own CSS architecture, rewriting it in utilities may cost more than it saves. Consistency inside one codebase matters more than any single methodology.
- **Tiny one-off pages.** A single static page with a dozen rules does not need a build step. A short hand-written stylesheet is perfectly fine.

For the kind of project this book is about — a marketing site plus an application dashboard, built by a small team, changing every week — utility-first CSS is an excellent fit. That is exactly the situation Tideline is in.

How Tailwind CSS v4 Generates Only What You Use

A natural worry at this point: if there are utilities for every spacing step, every color shade, and every combination of variants, the CSS file must be enormous. It is not, because Tailwind does not ship a fixed stylesheet at all. It generates one, on demand, from your source files.

At a high level, a build works in three steps:

1. **Scan.** Tailwind looks through the files in your project — HTML, templates, JavaScript, and so on — for tokens that look like class names. In v4 this is automatic: you do not list your files anywhere. It skips files listed in `.gitignore`, the `node_modules` folder, binary files, CSS files, and lock files.
2. **Generate.** For every token that matches a utility (with or without variants), it generates the corresponding CSS rule. Tokens that do not match anything, such as ordinary words in your text, are ignored.
3. **Optimize.** The result is processed by Lightning CSS, a fast CSS transformer built into Tailwind v4, which handles nesting, vendor prefixes, and — when you ask for it — minification.

The output contains a theme layer with the CSS variables your utilities need, a base layer of browser-default resets called **Preflight**, and a utilities layer with only the classes it found. In our experiment, the CSS for the two cards above — including Preflight and the theme variables — fit comfortably in a single small file. We will look inside that file in detail in Chapter 2.

This scanning approach has one consequence that catches almost everyone once, so it is worth learning now: Tailwind reads your source files as plain text. It never runs your JavaScript.

status-badge.js (does not work)

```
// Tailwind never sees "bg-teal-50" or "bg-rose-50" in this file,
// so it never generates them.
const badge = `bg-${color}-50 text-${color}-700`;
```

status-badge.js (works)

```
// Complete class names appear in the source, so Tailwind finds them.
const styles = {
  "In progress": "bg-teal-50 text-teal-700",
  "At risk": "bg-rose-50 text-rose-700",
};
const badge = styles[status];
```

The rule is simple: **always write complete class names in your source code**. Build them by choosing between whole strings, never by gluing fragments together. We come back to this rule in Chapter 2's troubleshooting checklist and in Chapter 24, where we build component maps for React and Vue.

> **NOTE — IF YOU ARE COMING FROM V3**
>
> Tailwind CSS v3 needed a `tailwind.config.js` file with a `content` array listing the files to scan, and a stylesheet with three `@tailwind` directives. Version 4 is **CSS-first**: one line, `@import "tailwindcss";`, replaces the directives, source files are detected automatically, and customization happens in CSS with `@theme`. This book teaches the v4 way throughout; Chapter 27 covers migrating an existing v3 project.

Meet Tideline

Learning a tool on isolated examples only goes so far. To see how decisions play out across a real interface, we build one product from start to finish: **Tideline**, a fictional SaaS product described as "project and time tracking for small creative agencies." Its tagline is "Every project, right on time."

Tideline has two faces, and we build both.

The marketing site

The public website at `tideline.example` has to convince a busy agency owner to start a trial. Its landing page opens with the headline "Plan, track, and bill every project in one place." and two calls to action, "Start free trial" and "Book a demo". It presents six features — Project boards, Time tracking, One-click invoices, Client portal, Team workload, and Reports — followed by testimonials, a call-to-action band, and a footer. Alongside it we build a pricing page with three plans and a Monthly/Annual toggle, log-in, sign-up, and password-reset pages, a long-form changelog, and a friendly 404 page.

The application

Behind the login is the app that agency teams use every day. We follow one customer workspace, **Brightside Studio**, on the Team plan with eight seats, signed in as its owner, Amara Okafor. The dashboard has a sidebar with Dashboard, Projects, Time, Invoices, Clients, Team, Reports, and Settings; a row of key numbers; a weekly hours chart drawn in pure CSS; a recent-activity feed; and a

team workload list. There are full tables of projects and invoices, modal dialogs for creating and deleting projects, toast notifications, a user menu, and a settings page with profile, workspace, billing, and notification panels.

All of this uses the same sample data from chapter to chapter. Brightside Studio's six projects, from Website Redesign for Harborline Coffee to Email Templates, appear with the same IDs, budgets, and due dates wherever they show up, and "today" in the app is always Tuesday, March 24, 2026. By the end of the book the pieces fit together into one coherent product, which we review as a whole in Chapter 28.

The plan of the book

Part	Chapters	What we build
I — Foundations	1–5	The setup, core utilities, variants, and Tideline's brand tokens
II — Layout and Responsive Design	6–10	Boxes, flexbox, grid, breakpoints, and the complete landing page
III — Components	11–15	Buttons, badges, forms and auth pages, pricing, navigation, and tables
IV — Theming and Design Systems	16–19	Dark mode, custom CSS, typography, and a documented design system
V — The Application UI	20–24	The dashboard shell, overlays, motion, accessibility, and framework components
VI — Production	25–28	Maintainable markup, builds, upgrading, and the finished case study

Every chapter is hands-on. You build a real piece of Tideline, and you see the CSS that each new utility produces when it teaches something. When a chapter introduces a concept that a later chapter completes, we say so, so you always know where a thread continues.

The project folder

The finished project lives in a folder called `tideline-ui/`. In Chapter 2 we create it with just three files — a `package.json`, a CSS entry file at `src/styles.css`, and an `index.html` — and it grows chapter by chapter until it looks like this:

```
tideline-ui/
  package.json
  src/styles.css          the CSS entry (Chapter 2)
  src/theme.css           design tokens (Chapter 5)
  src/components.css      component classes (Chapter 17)
  src/js/                 small vanilla JavaScript files
  dist/styles.css         the generated CSS (never edited by hand)
  index.html  pricing.html  login.html  signup.html  ...
  app/dashboard.html  app/projects.html  app/invoices.html  app/settings.html
```