

WEB DEVELOPMENT COURSE

Tailwind CSS Intermediate/Advanced

Tailwind v4 in real projects: the engine, design tokens, plugins, advanced variants, frameworks, component patterns, tooling, migration from v3, accessibility and a themed component kit. Every chapter was built and measured.

10 Chapters

Philip Osztromok

Generated with Claude

TAILWIND CSS INTERMEDIATE/ADVANCED

Contents

Chapter 1	Inside the Engine	2
Chapter 2	Design Tokens	13
Chapter 3	Utilities and Plugins	25
Chapter 4	Advanced Variants	36
Chapter 5	Frameworks	45
Chapter 6	Component Patterns	54
Chapter 7	Tooling and Maintenance	66
Chapter 8	Migrating to v4	75
Chapter 9	Accessible, Global, Printable	87
Chapter 10	Capstone Kit	97

Tailwind CSS Intermediate/Advanced

Chapter 1 · Inside Tailwind v4: Engine, Scanning and Layers

Tailwind CSS Fundamentals taught you to write the classes. This course is about running Tailwind in real projects, and almost every real-project problem (a class that does nothing, a stylesheet that is bigger than expected, old CSS that refuses to be overridden) comes down to the same question: **what does Tailwind actually do between your markup and the CSS file?** This chapter opens the box. You will see what the scanner reads and what it skips, how to point it at the files it misses, why class names built in code vanish, what the generated file is made of, why a single unlayered rule beats every utility, and what the final optimisation step changes.

> **EVERY EXAMPLE HERE WAS RUN**
>
> Everything below was built with the Tailwind CLI 4.3.3 (`@tailwindcss/cli`) on Node 24.18.0 and Windows 11, in scratch projects outside this site. Computed styles were read from headless Chrome. The CLI pulled in Lightning CSS 1.32.0 and `@parcel/watcher` 2.5.1 (checked with `npm ls`). Not tested: macOS or Linux, watching a network drive, the Vite and PostCSS plugins (chapter 5), and any Tailwind version other than 4.3.3.

The Pipeline in One Picture

Tailwind v4 does four jobs, in order. Knowing which one is misbehaving is most of the skill of debugging it:

STEP	WHAT HAPPENS	WHERE IT GOES WRONG
1 · Scan	A scanner (the Rust package `@tailwindcss/oxide`) reads your project files as **plain text** and pulls out every word that *might* be a class name	The file is never read (ignored, or in a folder it does not know about)
2 · Match	The compiler checks each candidate against the utilities and variants it knows, plus your theme. Real classes become rules; everything else is dropped silently	The class is not written out in full, or is misspelt (no error, just no rule)
3 · Arrange	The rules are placed in cascade layers, and the theme variables that were used are written out	Other CSS is outside the layers and wins

STEP	WHAT HAPPENS	WHERE IT GOES WRONG
4 · Transform	Lightning CSS (a Rust CSS tool) post-processes the result; with `--optimize` or `--minify` it also rewrites and compresses it	Surprise differences between dev and production output

The most important consequence is in step 1. **The scanner does not parse JavaScript, HTML or Markdown; it only looks for words.** It cannot tell a class attribute from a sentence. I proved that with a Markdown file that contains no markup at all:

```
We need to flex a little and keep the block hidden from view; the table is a fixed size.
```

```
.fixed   .block   .flex   .hidden   .table      (the five rules generated from that one
```

Five utilities were generated from a sentence. That is harmless, apart from a few extra bytes, but it explains two things you will see later: classes in comments and strings count, and a class name *assembled* from parts is invisible, because the finished word never appears in any file.

What the Scanner Reads, and What It Skips

By default v4 needs no `content` list. It scans the project from the current directory and skips what is clearly not source. To see exactly where the line falls, I put one distinct colour class in each kind of file, built, and checked which colours came out:

FILE (AND THE CLASS INSIDE IT)	USED?	WHY
`src/index.html` (`bg-red-500`)	yes	an ordinary source file
`src/page.md` (`bg-orange-500`)	yes	any text file counts, not just HTML and JS
`docs/notes.txt` (`bg-teal-500`)	yes	same
`dist/built.html` (`bg-yellow-500`)	no	`dist/` is listed in `.gitignore`
`private/secret.html` (`bg-green-500`)	no	also in `.gitignore`
`node_modules/pkg/index.js` (`bg-lime-500`)	no	`node_modules` is always skipped
`src/styles.css` (`.bg-cyan-500 {}`)	no	CSS files are never scanned

FILE (AND THE CLASS INSIDE IT)	USED?	WHY
`assets/logo.png` (text `bg-sky-500` inside)	no	binary file types are skipped
`package-lock.json` (`bg-blue-500`)	no	lock files are skipped

One detail surprised me, so I tested it twice: **the `.gitignore` is honoured even when the folder is not a git repository.** I ran the build before and after `git init` and the answer was identical (red, orange and teal only). The consequence is practical: a class that lives only in a generated or ignored folder will quietly not exist.

Telling the Scanner Where to Look

Three tools in your CSS file change what is scanned. All of them were run against the same nine files above, and the colours that appeared are in the last column.

DIRECTIVE	WHAT IT DOES	COLOURS THAT APPEARED
`@source "./node_modules/pkg";` `@source "./private";` `@source "./dist";`	Adds a folder to the automatic scan. It **overrides** `.gitignore` and the `node_modules` rule for that folder	red orange teal + lime, green, yellow
`@source not "./src/page.md";` `@source not "./docs";`	Removes a file or folder from the scan	red only
`@import "tailwindcss" source(none);` `@source "./src";`	Turns the automatic scan off and lists the folders yourself	red orange (the two scannable files in `src/`; `styles.css` still ignored)
`@import "tailwindcss" source("./docs");`	Moves the base directory of the automatic scan to `./docs`	teal only
`@source "./src/*.html";`	A glob: only matching files	red only

Two more rules from my runs. First, **paths in `@source` are relative to the CSS file that contains them**: with the CSS file in `css/`, I needed `@source "../src"` and it found red and orange. Second, naming a folder does not make Tailwind read file types it never reads: even with `@source "./src"`, the class in `src/styles.css` never appeared.

WHEN TO USE WHICH

Leave the automatic scan on for ordinary apps. Add `@source` for a component library installed in `node_modules` that ships Tailwind classes, or for a sibling package in a monorepo. Use `source(none)` only when you want full control (a shared theme package that scans nothing of its own). `@source not` is for generated folders that are not in `.gitignore`, or noisy content such as a CMS export full of prose.

Classes the Scanner Cannot See

This is the failure that costs the most time. Here is a badge function that builds its colour from an argument, and the check I ran on the output:

```
export const badge = (color) => `rounded px-2 py-1 text-white bg-${color}-600`;
badge("red");   // renders: bg-red-600
```

```
bg-red-600 / bg-green-600 / bg-blue-600 present in the CSS:   none
utility rules generated: 4   (rounded, px-2, py-1, text-white)
```

The browser receives `bg-red-600`, but the scanner only ever sees the file, where the text is `bg-${color}-600`, and that is not a class. The same goes for `"text-" + "red-600"`. Nothing warns you. There are three fixes, in order of preference:

PATTERN	HOW	RESULT IN MY RUN
Write the full name somewhere (best)	A lookup table: `{ red: "bg-red-600", green: "bg-green-600", blue: "bg-blue-600" }[color]`	all three colours present (7 rules)
Declare the set in CSS	`@source inline("bg-{red,green,blue}-600");` in your CSS	all three present (7 rules)
Do not use dynamic names at all	Pass the whole class as data (from your CMS, config or a prop), so it appears in a file the scanner reads	depends on where the data lives

`@source inline()` is the v4 replacement for the old `safelist` option, and it understands brace expansion, number ranges and variants. This single line, built and listed, produced all of the classes below:

```
@import "tailwindcss" source(none);
@source inline("underline");
@source inline("{hover:,focus:,}bg-red-{50,{100..900..100},950}");
@source inline("lg:text-{sm,lg}");
@source not inline("bg-red-{100,200}");
```

```
plain:   .underline  .bg-red-50  .bg-red-300 … .bg-red-900  .bg-red-950      (bg-red-100
focus:   .focus:bg-red-50 … .focus:bg-red-950   (11 rules, including 100 and 200)
hover:   .hover:bg-red-50 … .hover:bg-red-950   (inside @media (hover: hover), 11 rules)
lg:      .lg:text-sm  .lg:text-lg               (inside @media (width >= 64rem))
```

Look at the last line of the input. `@source not inline("bg-red-{100,200}")` removed exactly the two *plain* classes it named, and the `hover:` and `focus:` versions of the same colours stayed. It matches whole class names, so to remove the variants you must list them as well. The braces generate every combination, so keep the sets small: `@source inline("{hover:,focus:,}bg-red-{50,{100..900..100},950}")` is 33 classes from one line, and a careless range can add thousands.

Reading the Output

A small page (one button with six classes) builds to 188 lines and 5,191 bytes. Almost all of that is the same every time. The structure is what to read:

```
/*! tailwindcss v4.3.3 | MIT License | https://tailwindcss.com */
@layer theme, base, components, utilities;
@layer theme {
  :root, :host { --font-sans: …; --color-sky-600: oklch(58.8% 0.158 241.966); --spacing:
}
@layer base {
  *, ::after, ::before, ::backdrop, ::file-selector-button { box-sizing: border-box; mar
  …
}
@layer utilities {
  .rounded-lg { border-radius: var(--radius-lg); }
  .bg-sky-600 { background-color: var(--color-sky-600); }
  .px-4 { padding-inline: calc(var(--spacing) * 4); }
  …
```

```
  @media (hover: hover) { .hover\:bg-sky-700:hover { background-color: var(--color-sky-7
}
```

PART	CONTENTS	NOTES
`@layer theme, base, components, utilities;`	The order of the four layers, declared once, first	Later layers win over earlier ones
`theme`	Your design tokens as CSS variables on `:root`	Only the ones in use (see below)
`base`	The reset (Preflight): borders, margins, form controls, headings	Same on every build
`components`	Empty unless you write rules into it	Where plugins and your own component classes go
`utilities`	One rule per class found, in a fixed order,	The part that grows with your

Theme variables are written only when something uses them. I defined three tokens and used one:

```
@theme {
  --color-brand: #0af;
  --color-unused: #f0a;
}
@theme static {
  --color-always: #0fa;
}
```

```
With <div class="bg-brand"> in the page:   --color-brand: #0af;   --color-always: #0fa;
Without any use of bg-brand:                                      --color-always: #0fa;
```

Plain `@theme` keeps your file small; `@theme static` (or the `static` option) forces a variable out even when nothing uses it, which you need when other CSS or JavaScript reads it with `var(--color-always)` or `getComputedStyle`. Chapter 2 (*Tailwind CSS Intermediate/Advanced 2*) builds a whole token system on this.

Layers and the Cascade: Why Plain CSS Beats Utilities

Cascade layers change the order in which rules compete. The relevant fact, which is easy to miss because layers are new to many developers: **CSS that is not inside any layer beats CSS that is inside a layer, whatever the specificity**. Tailwind puts every utility inside `@layer utilities`, so any ordinary rule you write at the top level of a stylesheet outranks them. I tested four placements of an identical `font-size: 10px` rule against `text-xl` (20px) on the same elements, and read the result from Chrome:

```
.card1 { font-size: 10px; }                                  /* no layer */
@layer components { .card2 { font-size: 10px; } }
@layer utilities  { .card3 { font-size: 10px; } }           /* written after text-xl */
@layer base       { .card4 { font-size: 10px; } }

<p class="text-xl card1">  …  class="text-xl card2"  …  class="text-xl card3"  …  class=
```

```
card1 (unlayered)              font-size 10px     the plain rule wins
card2 (components layer)       font-size 20px     text-xl wins: utilities come after com
card3 (utilities layer, later) font-size 10px     same layer, same specificity, so the l
card4 (base layer)             font-size 20px     text-xl wins
```

That gives the working rules for any stylesheet in a Tailwind project:

- A rule in `@layer components` can be overridden by a utility on the element, which is the behaviour you want from a component class. It is why Tailwind provides that layer.
- An unlayered rule beats a utility. This is the usual explanation when "my `p-8` does nothing" on an element that an older stylesheet also styles.
- Inside one layer, the normal rules apply: higher specificity wins, then the later rule.

The clean fix for old CSS is not to raise specificity or add `!important`, but to put the old stylesheet into a layer. A legacy rule `.btn { padding: 4px; background-color: gray; }` used on `class="btn p-8 bg-blue-600"`:

```
@import "tailwindcss" source(none);
@source "./src";
@import "./src/legacy.css";                    /* version A: unlayered */
```

```
@import "./src/legacy.css" layer(components);     /* version B: into the components laye
```

```
A:  padding 4px    background rgb(128, 128, 128)          the legacy rule beats p-8 and
B:  padding 32px   background oklch(0.546 0.245 262.881)  the utilities win
```

A NOTE ON !IMPORTANT

Tailwind has an important modifier (a trailing `!`, as in `p-8!`) for one-off emergencies. It works (not run here), but it is a patch. Layering the old CSS fixes the cause, once, for every rule in it.

What Lightning CSS Does

The last step runs your CSS through Lightning CSS. The default build leaves most of your own CSS alone; `--optimize` and `--minify` rewrite it. I gave the CLI a rule that uses nesting, a prefixed property and `color-mix()`, and built it three ways:

```
.card {
  user-select: none;
  background: color-mix(in srgb, red 40%, blue);
  &:hover { color: red; }
  .title { font-weight: 700; }
}
```

```
default      5,524 bytes    kept exactly as written (nesting, color-mix, no prefix)

--optimize   6,238 bytes    .card { -webkit-user-select: none; user-select: none; backgr
                            .card:hover { color: red; }
                            .card .title { font-weight: 700; }

--minify     4,881 bytes    …}.card{-webkit-user-select:none;user-select:none;background
```

Three observations from that run, each worth knowing before you debug a production-only difference:

- **The nesting was flattened** (to `.card:hover` and `.card .title`), **the vendor prefix was added** (`-webkit-user-select`), and `color-mix()` **with two fixed colours was computed**

(`#609`). The tool lowers your CSS for the browsers it targets.

- `--optimize` **was *bigger* than the default build, not smaller.** I expected the opposite. Flattening and splitting grouped selectors into separate rules adds text; only `--minify` shrinks the file, and it also does the optimising.
- **The layer-order line disappears in optimised output** (`@layer theme, base, components, utilities;` is gone), but the order is kept: the layers still appear in sequence, with an empty `@layer components;` placeholder holding the third position.

So the rule is: compare like with like. If a bug shows only in production, build the development CSS with `--minify` locally and test that.

Watch Mode

`--watch` keeps the compiler running and rebuilds when a scanned file changes. I started it on a project with one file, then edited a class, then added a new file. The file lists after each step:

```
npx @tailwindcss/cli -i tw.css -o wout.css --watch=always

(1) start                          →  .p-1
(2) edit index.html: add underline →  .p-1  .underline
(3) create new.html with p-9       →  .p-1  .p-9  .underline
```

```
≈ tailwindcss v4.3.3
Done in 466ms      the first build
Done in 10ms
Done in 77ms
Done in 10ms
Done in 52ms       five builds in all: one at the start, then the ones triggered by my e
```

New files are picked up without restarting. Plain `--watch` stops when standard input closes (which happens under some task runners and CI jobs); `--watch=always` keeps going. The option list also has `--poll` for file systems that do not report changes. I did not test polling.

MISTAKES WHILE PREPARING THIS CHAPTER

Wrong assumption 1: "vendor code is skipped." My first test project put a library in `vendor/` and expected `@source` to be needed for it. It was scanned by default; only `node_modules`, ignored folders, binary files, CSS and lock files are skipped. I moved the library into `node_modules` to make the

exercise honest. **How to avoid it:** build with no directives first and look at what is already there before adding any.

Wrong assumption 2: "`.gitignore` needs a git repository." It does not; the ignore file is read anyway (tested before and after `git init`). **How to avoid it:** test the thing you remember, not the thing you assume.

Wrong assumption 3: "`--optimize` makes the file smaller." It made it 13% bigger (6,238 against 5,524 bytes). **How to avoid it:** measure with the flag you actually ship.

Hands-On Exercises

EXERCISE 1

Create a project with six files, each holding a different colour class: an HTML file in `src/`, a Markdown post in `content/`, a script in `node_modules/ui-kit/`, an HTML file in a `dist/` folder that your `.gitignore` lists, a CSS file and a PNG. Predict which of six colours will be generated, then build and check. Then make the `node_modules` script count, remove the Markdown post, and finally switch to `source(none)` with two folders listed.

View solution

EXERCISE 2

Write a `badge(color)` function that builds `bg-${color}-600`. Prove, from the generated CSS, that none of the three colours exist. Fix it two ways (a lookup table and `@source inline()`) and compare the number of rules generated each time.

View solution

EXERCISE 3

An old stylesheet defines `.btn { padding: 4px; background-color: gray; }`. Use it together with `p-8 bg-blue-600` on one button and read the computed padding and background in Chrome. Then fix it by importing the old file into the components layer, and read the computed values again. Explain each result in terms of layers.

View solution

CHAPTER 1 QUICK REFERENCE

- Pipeline: scan (plain text, Rust) → match against known utilities → arrange in layers → Lightning CSS. Unknown words are dropped silently, and prose words such as `flex`, `block` and `table` become real rules
- Skipped by default: `.gitignore`d paths (even without git), `node_modules`, binary files, CSS files, lock files. Scanned: any other text file, including `.md` and `.txt`
- `@source "./path"` adds (and overrides ignore rules); `@source not "./path"` removes; paths are relative to the CSS file; globs work
- `@import "tailwindcss" source(none)` turns the automatic scan off; `source("./dir")` moves its base directory
- Dynamic names such as `bg-${color}-600` never appear in any file, so they generate nothing. Fix: a lookup table of full names, or `@source inline("bg-{red,green,blue}-600")`
- `@source inline()` supports braces, ranges (`{100..900..100}`) and variants; `@source not inline()` removes only the exact names you list
- Output order: `@layer theme, base, components, utilities;`; theme variables appear only when used (`@theme static` forces them)
- Unlayered CSS beats every layered rule; `@layer components` loses to utilities; import old CSS with `layer(components)`
- `--optimize` flattens nesting, adds prefixes and folds constants but can be **larger**; `--minify` is the one that shrinks. Test production CSS with `--minify`
- `--watch` rebuilds in tens of milliseconds and sees new files; `--watch=always` survives closed stdin

COMING NEXT

Chapter 2 builds on the theme variables you just saw written to the output: semantic design tokens (`--color-surface`, `--color-primary`), `@theme inline`, switching between brand and light/dark themes with a `data-theme` attribute, and modern colour with `oklch` and `color-mix()`. Start with *Tailwind CSS Intermediate/Advanced 2*.

Tailwind CSS Intermediate/Advanced

Chapter 2 · Design Tokens and Multi-Theme Systems

Writing `bg-blue-600` says what a button *looks like*. Writing `bg-primary` says what it is *for*, and that small change is what lets one codebase carry a light theme, a dark theme and several brand colours without a single `dark:` class in the markup. This chapter builds that system: tokens with meaning, the one Tailwind feature (`@theme inline`) that makes it work, themes switched by an HTML attribute, modern colour (`oklch`, `color-mix()`, wide gamut) and the scales for spacing, type and radius. It builds on *Tailwind CSS Fundamentals 9* (`@theme` basics) and *Tailwind CSS Fundamentals 8* (dark mode with `dark:`).

> **EVERY EXAMPLE HERE WAS RUN**
>
> Tailwind CLI 4.3.3, Node 24.18.0 and headless Chrome 154 on Windows 11. Colours were read from the browser: contrast ratios were computed in the page from real pixels (a canvas read in sRGB), and "prefers dark" was set with Chrome's `--force-dark-mode` flag. Not tested: Firefox and Safari, a real wide-gamut (P3) display, and a screen reader. Where a result depends on those, the text says so.

Palette Tokens and Semantic Tokens

Tailwind ships a **palette**: `--color-blue-600`, `--color-gray-100`, and so on. Those names describe a colour. A design system adds a second layer of names that describe a *role*. The two layers answer different questions:

	PALETTE TOKEN	SEMANTIC TOKEN
Example names	`blue-600`, `gray-100`, `red-500`	`primary`, `surface`, `ink`, `muted`, `danger`
Answers	What colour is it?	What is it for?
Changing the brand	edit every class that uses blue-600	edit one variable
Dark mode	a `dark:` class on each element	nothing in the markup: the variable changes
Good for	one-off pages, prototypes, a decorative colour	anything shared: buttons, inputs, cards, navigation

The plan for the rest of the chapter: **palette tokens stay as they are, and a small set of semantic tokens points at them**, with the pointing changed per theme. For the pointing to work in Tailwind v4 you need one detail that looks like a stylistic choice and is not.

Why `@theme inline` Exists

The obvious way to make a semantic token is to write it in `@theme` and point it at a variable, then change that variable in a themed section of the page. I built exactly that and put the same `bg-primary` element in two places, one inside `data-theme="ocean"`:

```
@import "tailwindcss" source(none);
@source "./tpl.html";
:root { --brand: #2563eb; }
[data-theme="ocean"] { --brand: #0d9488; }
@theme {
  --color-primary: var(--brand);
}

/* b.css: identical, except   @theme inline { --color-primary: var(--brand); }   */
```

```
plain @theme
  root:                       rgb(37, 99, 235)    blue
  inside data-theme=ocean:    rgb(37, 99, 235)    still blue: the theme did nothing
  generated CSS:              --color-primary: var(--brand);          (on :root)
                              .bg-primary { background-color: var(--color-primary); }

@theme inline
  root:                       rgb(37, 99, 235)
  inside data-theme=ocean:    rgb(13, 148, 136)   teal: it worked
  generated CSS:              .bg-primary { background-color: var(--brand); }
```

A custom property that contains `var()` is resolved **where it is declared**. With plain `@theme`, Tailwind declares `--color-primary` on `:root`, where `--brand` is blue, and every element below simply inherits that finished colour; redefining `--brand` in the ocean section changes a variable that nothing re-reads. With `inline`, Tailwind does not create `--color-primary` at all; it copies your `var(--brand)` straight into the utility, so it is read on the element, at the nearest `--brand`.

USE	WHEN	COST
`@theme { … }`	The value is fixed (a palette colour, a size, a font)	none: the variable is written, and other CSS and JavaScript can read it
`@theme inline { … }`	The value is a `var()` that changes per scope (themes, brands, a component's local setting)	**the token variable is not written**: I counted zero occurrences of `--color-surface` in the output. Read `var(--surface)` in your own CSS and JS
`@theme static { … }`	You want the variable written even when no class uses it (chapter 1)	a few bytes per unused variable

A Complete Multi-Theme System

Here is the whole system in one file: six semantic tokens, a light default, a dark mode, and a second brand ("sunset"). Two attributes pick the theme, and they are independent of each other, so a page can be light or dark in either brand without writing the four combinations by hand:

```
@import "tailwindcss" source(none);
@source "./tpl.html";

@theme inline {
  --color-surface: var(--surface);
  --color-ink: var(--ink);
  --color-muted: var(--muted);
  --color-line: var(--line);
  --color-primary: var(--primary);
  --color-primary-ink: var(--primary-ink);
}

:root {
  --surface: oklch(0.99 0 0);
  --ink: oklch(0.25 0.02 260);
  --muted: oklch(0.5 0.02 260);
  --line: oklch(0.9 0.01 260);
  --primary: oklch(0.5 0.2 260);
  --primary-ink: oklch(1 0 0);
}
[data-mode="dark"] {
  --surface: oklch(0.2 0.02 260);
  --ink: oklch(0.95 0.01 260);
```

```
    --muted: oklch(0.7 0.02 260);
    --line: oklch(0.35 0.02 260);
    --primary: oklch(0.7 0.15 260);
    --primary-ink: oklch(0.2 0.02 260);
  }
  [data-brand="sunset"] { --primary: oklch(0.55 0.19 40); }
  [data-mode="dark"][data-brand="sunset"] { --primary: oklch(0.75 0.15 50); }
```

The markup uses only meaning, never a colour name or a `dark:`:

```
<div data-mode="dark" data-brand="sunset" class="bg-surface text-ink p-2">
  <button class="bg-primary text-primary-ink border border-line px-3 py-1">Save</button>
  <span class="text-muted">hint</span>
</div>
```

I put that block in four contexts (every mode and brand combination) and had the page measure the result. The contrast column is the WCAG ratio, computed in the page from the rendered colours; 4.5 is the AA minimum for normal text.

```
light / default   surface #fcfcfc  ink #1c222b (15.59)  muted (5.82)  primary #0559d2  t
dark  / default   surface #11161f  ink #ebeff5 (15.71)  muted (6.79)  primary #659dfb  t
light / sunset    surface #fcfcfc  ink #1c222b (15.59)  muted (5.82)  primary #c73800  t
dark  / sunset    surface #11161f  ink #ebeff5 (15.71)  muted (6.79)  primary #f88f4f  t
```

Every pair is above 4.5. Notice the rule `[data-mode="dark"][data-brand="sunset"]` in the CSS: it exists because the orange that is readable on a pale page (lightness 0.55) would be too dark on a dark one, so the dark version is lighter (0.75). The numbers are the proof, and they are the reason to keep a contrast check next to the tokens. Because the attribute decides which variables apply, changing it in JavaScript changes the theme with no rebuild: I set `data-brand="sunset"` on the root element from a script and read the new `--primary` straight back (`#0559d2` became `#c73800`).

Derived Colours with `color-mix()`

Hover and active colours should not be separate hand-picked tokens that you must remember to update in every theme. Derive them:

```
@theme inline {
  --color-primary: var(--primary);
  --color-primary-hover: var(--primary-hover);
}
:root {
  --primary: oklch(0.5 0.2 260);
  --primary-hover: color-mix(in oklab, var(--primary), black 15%);
}
[data-brand="sunset"] { --primary: oklch(0.55 0.19 40); }
```

```
default brand:   --primary #0559d2   --primary-hover #0346a9
sunset brand:    --primary #c73800   --primary-hover #9f2b00
```

The hover colour is recomputed from whichever `--primary` applies, so adding a brand never means adding a hover colour.

Following the Operating System: `light-dark()`

Tailwind CSS Fundamentals 8 showed how to follow `prefers-color-scheme` with the `dark:` variant. The token system has a tidier option: the CSS function `light-dark(a, b)`, which picks its first argument when the element's `color-scheme` is light and its second when dark. Define each token once, and let an attribute override the system choice:

```
@import "tailwindcss" source(none);
@source "./tpl.html";
@theme inline {
  --color-surface: var(--surface);
  --color-ink: var(--ink);
}
:root {
  color-scheme: light dark;
  --surface: light-dark(oklch(0.99 0 0), oklch(0.2 0.02 260));
  --ink: light-dark(oklch(0.25 0.02 260), oklch(0.95 0.01 260));
}
[data-mode="light"] { color-scheme: light; }
[data-mode="dark"]  { color-scheme: dark; }
```

```
Browser prefers light:                        Browser prefers dark (Chrome --force-dark-
  no attribute      surface #fcfcfc               no attribute      surface #11161f
```

```
  data-mode=light   surface #fcfcfc              data-mode=light   surface #fcfcfc
  data-mode=dark    surface #11161f              data-mode=dark    surface #11161f
```

With no attribute the page follows the system; an explicit `data-mode` wins either way, which is what a theme toggle needs (three states: system, light, dark). The `light-dark()` function needs `color-scheme` to be set on an ancestor, which is why `:root` declares `light dark`. Compared with duplicating a dark block inside `@media (prefers-color-scheme: dark)`, each token is written once with both values side by side. One of my own runs went wrong at this point and is in the "Mistakes" box. Setting `color-scheme` is also what tells the browser to draw its built-in controls and scrollbars in the matching scheme (documented behaviour; I did not look at the controls).

Colour That Scales: `oklch`

`oklch(L C H)` describes a colour by **lightness** (0 to 1), **chroma** (how vivid) and **hue** (an angle in degrees). Tailwind v4's own palette is written in it, which is why you saw values such as `oklch(58.8% 0.158 241.966)` in the generated file. For a theme system its great advantage is practical: **to make a new brand you keep lightness and chroma and change the hue.** I built a three-step scale on one variable, `--hue`:

```
@import "tailwindcss" source(none);
@source "./tpl.html";
@theme inline {
  --color-brand-100: oklch(0.95 0.04 var(--hue));
  --color-brand-500: oklch(0.62 0.19 var(--hue));
  --color-brand-900: oklch(0.30 0.08 var(--hue));
}
:root { --hue: 260; }
[data-brand="sunset"] { --hue: 45; }
[data-brand="forest"] { --hue: 150; }
```

```
<div class="bg-brand-500">                    oklch(0.62 0.19 260)     (hue from :root
<div data-brand="sunset" class="bg-brand-500">   oklch(0.62 0.19 45)
<div data-brand="forest" class="bg-brand-500">   oklch(0.62 0.19 150)
<div data-brand="sunset" class="bg-brand-500/20"> oklab(0.62 0.13435 0.13435 / 0.2)
```

This only works with `@theme inline`, for the reason in the previous section (the hue must be read on the element). Two details in the output matter. First, the opacity modifier (`/20`) works on a variable-based colour, and the browser resolves it to an `oklab` value with 0.2 alpha. Second, here is the rule Tailwind generated for it:

```
.bg-brand-500\/20 {
  background-color: oklch(0.62 0.19 var(--hue));
  @supports (color: color-mix(in lab, red, red)) {
    background-color: color-mix(in oklab, oklch(0.62 0.19 var(--hue)) 20%, transparent);
  }
}
```

The first declaration is the fallback for a browser without `color-mix()`, and it is the **full-strength colour, not a 20% one**. In a modern browser you will never see it (Chrome 154 reported support for `oklch`, `color-mix` and `light-dark`), but if you support older browsers, a translucent overlay can turn opaque there. Check your support target before relying on `/20` for anything that must stay see-through.

Wide-Gamut Colour and Its Fallbacks

`oklch` can describe colours that an ordinary sRGB screen cannot show; "wide gamut" screens (Display P3) can show more. I asked the browser what happens to an out-of-range colour on my (sRGB) setup by drawing it and reading the pixel back:

```
oklch(0.7 0.15 150)          ->  rgb(76,184,106)    inside sRGB
oklch(0.7 0.3 150)           ->  rgb(0,203,0)       outside sRGB: the red and blue chann
color(display-p3 0 1 0)      ->  rgb(0,255,0)       the P3 green, shown as the nearest s
@media (color-gamut: p3)     false normally;  true with Chrome's --force-color-profile=d
```

So out-of-range colours do not break; the browser maps them to the nearest displayable colour. Two practical rules follow:

- **Keep your base tokens inside sRGB** so every user sees the colour you chose. Test it the way I did: draw the colour and read it back; a channel stuck at exactly 0 or 255 (like the 0 and 0 above) means it was clipped.

- **Upgrade for wide-gamut screens on purpose**: put the vivid value in `@media (color-gamut: p3) { :root { --accent: oklch(0.7 0.3 150); } }`, so only displays that can show it receive it. I confirmed that the query becomes true under the P3 flag; I could not check how it looks, because I have no wide-gamut display and the canvas reads back sRGB only.

The build tool helps with one case. With `--optimize`, a `color(display-p3 …)` value gets an sRGB fallback line above it (`color: #00f942;` then `color: color(display-p3 0 1 0);`), but an `oklch()` value is left alone, and values inside custom properties are never rewritten. So the safety net is your own design rule, not the compiler.

Spacing, Type and Radius as Tokens

The same `@theme` mechanism that holds colour holds every scale. Each namespace creates utilities, and one line can move a whole family:

VARIABLE	CREATES	NOTES FROM MY BUILD
`--color-*`	bg-, text-, border-, ring-, fill-… for each name	`--color-*: initial;` removes the whole default palette first
`--spacing`	p-, m-, gap-, w-, h-, inset-… as `calc(var(--spacing) * n)`	set to `0.5rem`: `p-4` became 2rem. One line rescales every spacing utility
`--text-name`	`text-name` (font size)	companions in the same namespace: `--text-name--line-height`, `--text-name--font-weight`
`--radius-name`	`rounded-name`	`--radius-card: 0.75rem` gave `rounded-card`
`--font-name`	`font-name`	the default sans and mono stacks live here
`--*: initial`	clears everything first	only your own tokens remain: my output had two variables and no `rounded-lg` or `text-display`

```
@import "tailwindcss" source(none);
@source "./tpl.html";
@theme {
  --color-*: initial;
  --color-ink: oklch(0.25 0.02 260);
  --color-brand: oklch(0.55 0.2 260);
```

```
  --spacing: 0.5rem;
  --text-body: 1rem;
  --text-body--line-height: 1.6;
  --text-display: 2.5rem;
  --text-display--line-height: 1.1;
  --text-display--font-weight: 700;
  --radius-card: 0.75rem;
}
```

```
text-ink: rule     bg-brand: rule     p-4: rule        rounded-card: rule   text-body: r
text-display: rule bg-red-500: NO RULE  text-white: NO RULE   rounded-lg: rule

  .p-4 {
    padding: calc(var(--spacing) * 4);
  }
  .text-display {
    font-size: var(--text-display);
    line-height: var(--tw-leading, var(--text-display--line-height));
    font-weight: var(--tw-font-weight, var(--text-display--font-weight));
  }
```

Look at the second row of results: `text-white` **stopped working** after I wrote `--color-*: initial`, because white is part of the colour namespace. If you reset the palette, define `white`, `black` and `transparent` yourself if you still use them. Resetting is the right move for a strict design system (no one can reach for `bg-red-500`), and the wrong move for a project that mixes semantic and palette colours.

Documenting the System

A token set that nobody can see is a token set that people bypass. Two cheap checks keep the documentation honest. The first is a report built from Tailwind's own output, grouped by namespace; the one I wrote (about 15 lines of Node, in the exercise) printed this for the file above:

```
## spacing (1)   --spacing  0.5rem
## radius  (2)   --radius-lg 0.5rem   --radius-card 0.75rem
## color   (2)   --color-ink oklch(0.25 0.02 260)   --color-brand oklch(0.55 0.2 260)
## text    (5)   --text-body 1rem   --text-body--line-height 1.6   --text-display 2.5rem
```

Two honest limits. The report shows only tokens that the page *uses* (that is the rule from chapter 1), and it cannot list `@theme inline` tokens at all, since they are not written to the output. For those, read your source file. The second check is the contrast table from the multi-theme section: a small page that renders every theme and measures every text/background pair, run before you ship a new brand colour. I ran it by hand; wiring it into continuous integration is up to you and is *not run here*.

MISTAKES WHILE PREPARING THIS CHAPTER

1. I overwrote my own test page. My loop ran `sed "s/OUT/$v.out.css/" a.html > a.html` for the first variant: redirecting a command's output to the file it is reading empties the file first. The page was blank and the second variant scanned an empty file. **How to avoid it:** keep a template (`tpl.html`) and write the result to a different name.

2. I assumed "prefers dark" was on when it was not. The first flag I tried (`--blink-settings=preferredColorScheme=1`) silently did nothing, and the output for "dark preference" was identical to the light run. Only a line I had added to the page (`prefers dark: false`) exposed it. **How to avoid it:** make the test print its own precondition. `--force-dark-mode` worked.

3. I expected `--color-primary` to exist under `@theme inline`. It does not, so my token-report idea would not have listed any semantic token. **How to avoid it:** open the generated file before writing tooling that depends on its contents.

4. I ran `chrome --version`, which does not print a version here: it handed the command to the Chrome window already open on my machine. Nothing was changed, but it is not a harmless query. I read the version from the user-agent string in the headless page instead (Chrome 154).

Hands-On Exercises

EXERCISE 1

Take the six-token system from this chapter and build the page with the four contexts. Read back, for each one, the surface, ink and primary colours and the contrast of the ink on the surface and of the text on the primary button. Then change the sunset light-mode primary so that its contrast on the button text falls below 4.5, and show that your check catches it.

View solution

EXERCISE 2

Define `--color-primary: var(--brand)` with plain `@theme` and then with `@theme inline`. Put one `bg-primary` element at the root and one inside a section that sets a different `--brand`. Read both backgrounds each time, find the relevant lines in the generated CSS, and explain the difference.

View solution

EXERCISE 3

Write an `@theme` block that resets the palette, defines two colours, sets `--spacing` to 0.5rem, adds a `body` and a `display` text size with line heights and a `card` radius. Write a short Node script that lists the variables Tailwind wrote, grouped by namespace, and check which of nine classes (including `bg-red-500` and `text-white`) still have a rule.

View solution

CHAPTER 2 QUICK REFERENCE

- Palette tokens say what a colour *is*; semantic tokens (`primary`, `surface`, `ink`, `muted`, `line`) say what it is *for*. Markup uses the semantic ones
- `@theme inline { --color-primary: var(--brand); }`: the utility reads `var(--brand)` on the element. Plain `@theme` resolves at `:root` and ignores per-scope overrides
- Inline tokens are **not written** to the output; read the source variable (`var(--brand)`) in your own CSS and JavaScript
- Themes: re-define the source variables under attribute selectors (`[data-mode="dark"]`, `[data-brand="sunset"]`); two independent attributes cover mode × brand; add a combined selector only where a pair needs its own value
- `light-dark(a, b)` with `color-scheme: light dark` on `:root` follows the system; `[data-mode="light|dark"] { color-scheme: … }` overrides it
- `oklch(L C H)`: change the hue variable for a new brand; keep chroma inside sRGB for base tokens; `color-mix(in oklab, var(--primary), black 15%)` derives hover colours
- `/20` opacity on a variable colour becomes `color-mix()`; the no-support fallback is the *opaque* colour
- Wide gamut: out-of-range colours are mapped to the nearest displayable one; upgrade deliberately inside `@media (color-gamut: p3)`

- Scales: `--spacing` (base unit), `--text-x` (+ `--line-height`, `--font-weight`), `--radius-x`, `--font-x`; `--color-*: initial` removes the palette (and `white`); `--*: initial` removes everything
- Document with a token report from the compiled theme layer plus a contrast check across every theme

COMING NEXT

Chapter 3 goes beyond tokens to your own utilities and variants: `@utility` (static and functional), `@custom-variant`, compound variants, `@plugin` with the official typography and forms plugins, writing a small plugin, and `@config` for older JavaScript configs. Start with *Tailwind CSS Intermediate/Advanced 3*.

Tailwind CSS Intermediate/Advanced

Chapter 3 · Custom Utilities, Variants and Plugins

Sooner or later you want a class Tailwind does not have: `content-auto`, a `stack-3` that follows your spacing scale, a `hocus:` variant for hover-or-keyboard-focus, a `prose` class for rendered Markdown. Tailwind v4 gives you five ways to add one, and picking the wrong one is the usual mistake. This chapter builds each one, shows exactly what it generates, and ends with a rule for choosing. It assumes *Tailwind CSS Fundamentals 10* (arbitrary values, `@layer` and `@apply`) and *Tailwind CSS Intermediate/Advanced 1* (how the layers work).

EVERY EXAMPLE HERE WAS RUN

Tailwind CLI 4.3.3 on Node 24.18.0, with `@tailwindcss/typography` 0.5.20 and `@tailwindcss/forms` 0.5.11, and headless Chrome 154 for computed styles. Not tested: other Tailwind versions, plugins published by other authors, using a plugin through the Vite or PostCSS integrations (chapter 5), and any browser but Chrome.

Which Mechanism Do I Need?

Start from what you are trying to add, not from the feature name:

I WANT TO…	USE	WHY
Style one element once	a normal class, or an arbitrary value (`w-[13px]`)	no new name needed
Add a fixed-purpose class (`content-auto`)	`@utility name { … }` (static)	works with every variant, lives in the utilities layer
Add a class that takes a value (`tab-4`, `stack-3`)	`@utility name-* { … --value() … }` (functional)	values come from your theme or the class itself, and arbitrary `[…]` values work
Add a new condition (`hocus:`, `theme-midnight:`)	`@custom-variant`	combines with every utility and every other variant
Use Tailwind's variants inside my own CSS rule	`@variant`	hover, md and dark without writing the media queries

I WANT TO...	USE	WHY
Use a packaged set of styles (typography, forms)	`@plugin`	one line, with options
Read options, generate many rules, share through npm	a JavaScript plugin (`plugin()` or `plugin.withOptions()`)	the full API
Keep an old project's `tailwind.config.js` working	`@config`	a bridge, not a destination
Name a reusable chunk of UI	a component in your framework (*Tailwind CSS Fundamentals 12*)	not a Tailwind feature at all

Static Utilities: `@utility`

The simplest case is a name and some declarations. It lands in the utilities layer, so every variant works on it for free. I wrote two and used them with `hover:` and `lg:`:

```
@utility content-auto { content-visibility: auto; }

@utility scrollbar-hidden {
  scrollbar-width: none;
  &::-webkit-scrollbar { display: none; }
}

<i class="content-auto hover:content-auto lg:scrollbar-hidden"></i>
```

```
.content-auto { content-visibility: auto; }
@media (hover: hover) { .hover\:content-auto:hover { content-visibility: auto; } }
@media (width >= 64rem) {
  .lg\:scrollbar-hidden { scrollbar-width: none; }
  .lg\:scrollbar-hidden::-webkit-scrollbar { display: none; }
}
```

Two things to know. First, nesting (`&::-webkit-scrollbar`) is how one utility writes more than one rule. Second, **a name that already exists is merged, not replaced**. I defined `@utility underline { text-decoration-line: line-through; }` and the output was a single `.underline` rule containing *both* declarations, `underline` first and `line-through` second, so mine won. That is occasionally useful and usually an accident. A custom utility can also be used in your own

CSS: `.card { @apply glow-brand p-4; }` compiled to a plain rule with both utilities' declarations.

Functional Utilities: `--value()` and `--modifier()`

End a name with `-*` and the utility takes a value. The `--value()` function says what values are legal, and **a class whose value does not match is simply not generated**:

FORM	EXAMPLE	MATCHES
Theme key	`--value(--tab-size-*)`	names defined in `@theme` (`--tab-size-github: 8` gives `tab-github`)
Bare type	`--value(integer)`	`tab-4`. Types: `number`, `integer`, `ratio`, `percentage`
Arbitrary	`--value([integer])`	`tab-[12]`; other types: `length`, `color`, `url`, `*`…
Literal	`--value("inherit", "initial")`	the exact words
Spacing scale	`--spacing(--value(integer))`	turns `3` into `calc(var(--spacing) * 3)`
Modifier	`--modifier(integer)`	the part after a slash: `pad-x-4/7`; left out when absent
Default	`--value(integer, --default(4))`	makes the bare class (`tab`) work

```
@theme { --tab-size-github: 8; --color-brand: oklch(0.55 0.2 260); }

@utility tab-*   { tab-size: --value(--tab-size-*);  tab-size: --value(integer);  tab-si
@utility stack-* { display: flex; flex-direction: column;  gap: --spacing(--value(intege
@utility glow-*  { box-shadow: 0 0 0 3px --value(--color-*, [color]); }
@utility pad-x-* { padding-inline: --spacing(--value(integer));  line-height: --modifier
@utility nudge-*  { translate: --spacing(--value(integer)) 0; }
@utility -nudge-* { translate: calc(--spacing(--value(integer)) * -1) 0; }

<i class="tab-4 tab-github tab-[12] tab-x tab-4.5"></i>
<i class="stack-3 stack-[10px] stack-[2rem] stack-x"></i>
<i class="glow-brand glow-[#f00] glow-red-500 glow-nope"></i>
```

```
<i class="pad-x-4 pad-x-4/7"></i>
<i class="nudge-2 -nudge-2"></i>
```

```
generated:   tab-4  tab-github  tab-[12]  stack-3  stack-[10px]  stack-[2rem]  glow-bran
             pad-x-4  pad-x-4/7 (line-height: 7)  nudge-2  -nudge-2
not written: tab-x  tab-4.5  stack-x  glow-nope        (no error, no rule)

.stack-3 { display: flex; flex-direction: column; gap: calc(var(--spacing) * 3); }
.glow-brand { box-shadow: 0 0 0 3px var(--color-brand); }   (reads the theme variable, s
```

Notice that several `--value()` declarations in one rule are tried in turn and the ones that fail are dropped; that is how `stack-3` and `stack-[10px]` both work from one utility. Notice also that the negative version, `-nudge-2`, needed its own `@utility -nudge-*`: a negative is never derived for you.

A MISTAKE THAT DOES NOT FAIL THE BUILD

My first `glow-*` was `box-shadow: 0 0 0 --value([length], 3px) …`, trying to give the width a default. A literal such as `3px` is not allowed in `--value()` (bare values must be a number, integer, ratio or percentage). Tailwind printed `Unsupported bare value data type: "3px"` with a pointer to the exact spot, and then **exited with code 0 and wrote the CSS file anyway**, with the one utility missing and every other utility present. I measured the exit code without a pipe: 0. A CI step that checks only the exit code would pass. Search the output for the error text, or assert that a known class exists in the result.

Custom Variants: `@custom-variant`

A variant is a condition wrapped around a utility. The short form takes a selector in brackets; the block form wraps `@slot` (where the utility's declarations go) in whatever you need:

```
@import "tailwindcss" source(none);

@custom-variant theme-midnight (&:where([data-theme="midnight"], [data-theme="midnight"]
@custom-variant hocus (&:is(:hover, :focus-visible));
@custom-variant any-hover {
  @media (any-hover: hover) {
    &:hover { @slot; }
  }
```

```
}
@custom-variant desktop-hover {
  @media (width >= 64rem) {
    @media (hover: hover) {
      &:hover { @slot; }
    }
  }
}
@custom-variant dark-hover {
  @variant dark {
    @variant hover {
      @slot;
    }
  }
}
```

```
theme-midnight:bg-black    .…:where([data-theme="midnight"], [data-theme="midnight"] *) {
hocus:underline            .…:is(:hover, :focus-visible) { text-decoration-line: underlin
any-hover:font-bold        @media (any-hover: hover) { .…:hover { font-weight: … } }
desktop-hover:italic       @media (width >= 64rem) { @media (hover: hover) { .…:hover { f
dark-hover:uppercase       @media (prefers-color-scheme: dark) { @media (hover: hover) {

md:theme-midnight:p-2      inside @media (width >= 48rem), with the midnight selector
theme-midnight:hover:text-white   the midnight selector, plus the hover media query
```

What the output shows:

- `:where()` **keeps specificity at zero.** The `theme-midnight` selector is wrapped in it, so the variant never makes a utility harder to override than the same utility without the variant.
- **Conditions nest.** `desktop-hover` needs a wide screen *and* a hover-capable device, and two nested `@media` blocks say exactly that.
- **You can build a variant out of existing ones.** `dark-hover` uses `@variant dark` and `@variant hover` inside `@custom-variant`, and the output contains both conditions. (The course outline called this "compound @variant, v4.3". The documentation page does not use that term, so I tested the behaviour rather than the name; it works in 4.3.3 and I did not check which release first allowed it.)
- **Everything combines.** `md:theme-midnight:p-2` and `theme-midnight:hover:text-white` work, and `group-hocus:` appeared with no extra work.

Tailwind's Variants in Your Own CSS: `@variant`

Use `@variant` inside a rule you write yourself (for example a legacy class) to get Tailwind's variants without repeating the media queries:

```
.btn {
  background: white;
  @variant hover { background: black; }
  @variant md { padding: 1rem; }
  @variant hover:focus { outline: 2px solid red; }
  @variant dark, print { color: gray; }
}
```

```
.btn {
  background: white;
  &:hover { @media (hover: hover) { background: black; } }
  @media (width >= 48rem) { padding: 1rem; }
  &:hover { @media (hover: hover) { &:focus { outline: 2px solid red; } } }
  @media (prefers-color-scheme: dark) { color: gray; }
  @media print { color: gray; }
}
```

Stacking (`hover:focus`) nests the conditions and a comma list repeats the declarations for each variant. One caution from chapter 1: this `.btn` rule is **outside every layer** (the output shows it after the `@layer` blocks), so it beats utilities such as `p-8` on the same element. Wrap it in `@layer components` if you want utilities to win.

Official Plugins: Typography and Forms

A plugin is added with one line. Both official plugins declare Tailwind 4 in their `peerDependencies` and both worked in my build. Each adds rules, so I measured the cost and the effect. The baseline stylesheet (one page, no plugin) was 4,614 bytes:

```
@plugin "@tailwindcss/typography";                        /* the classes prose, prose-lg
@plugin "@tailwindcss/typography" { className: wysiwyg; } /* rename the class */
@plugin "@tailwindcss/forms";                             /* styles bare inputs, selects
@plugin "@tailwindcss/forms" { strategy: class; }         /* only .form-input, .form-sel
```

	size	<h2> in .prose	<p> in .prose	bar
no plugin	4,614 B	16px / 400	16px / 24px	bor
typography (class prose prose-lg)	27,741 B	30px / 700	18px / 32px	–
typography, className: wysiwyg	20,677 B	24px / 700	16px / 28px	–
forms	12,962 B	–	–	bor
forms, strategy: class	6,804 B	–	–	bor

Points from those numbers. **Typography** turned unstyled HTML (an `h2`, a paragraph, a link, a list) into styled content through one class; the link became underlined with weight 500. After I renamed the class to `wysiwyg`, the markup still said `prose prose-lg` and **neither did anything**: the heading was the base 24px, not the 30px of `prose-lg`. The class option replaces the name; it does not add an alias. **Forms** by default restyles every bare input in the project, which is what you want for a quick reset and not what you want if other CSS styles those inputs; `strategy: class` leaves plain inputs alone and only styles those with the class. The sizes depend on which of the plugin's classes your pages use, so treat them as one measurement, not a constant.

Writing a Plugin

The JavaScript plugin API is still the way to ship something other people install, or to generate many rules from data. A plugin file exports a function; `plugin.withOptions()` lets the CSS pass it options. Here is a small one that uses most of the helpers:

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

module.exports = plugin.withOptions(function (options = {}) {
  const color = options.color || 'rgb(0 0 0 / 0.3)';
  return function ({ addBase, addComponents, addUtilities, matchUtilities, addVariant, m
    addBase({ 'h1, h2': { 'text-wrap': 'balance' } });
    addComponents({ '.pill': { 'border-radius': '9999px', 'padding': '0.25rem 0.75rem' }
    addUtilities({ '.no-scrollbar': { 'scrollbar-width': 'none' } });
    matchUtilities(
      { 'glow-text': (value) => ({ 'text-shadow': `0 0 ${value} ${color}` }) },
      { values: { sm: '2px', md: '6px', lg: '12px' } }
    );
    matchUtilities(
      { 'accent-ring': (value) => ({ 'box-shadow': `0 0 0 3px ${value}` }) },
      { values: theme('colors') , type: 'color' }
```

```
    );
    addVariant('hocus', '&:is(:hover, :focus-visible)');
    matchVariant('nth', (value) => `&:nth-child(${value})`, { values: { 1: '1', 2: '2', (
  };
});

/* in the CSS: */
@theme { --color-brand: oklch(0.55 0.2 260); }
@plugin "./shadow.js" { color: rgb(255 0 0 / 0.5); }
```

The markup and what came out:

```
<h1 class="pill no-scrollbar glow-text-md glow-text-[9px] hocus:underline nth-odd:italic
```

```
addBase          base layer:      h1, h2 { text-wrap: balance; }
addUtilities     .no-scrollbar { scrollbar-width: none; }
matchUtilities   glow-text-md  → text-shadow: 0 0 6px rgb(255 0 0 / 0.5)      (the colou
                 glow-text-[9px] works without being listed
                 accent-ring-brand → box-shadow: 0 0 0 3px oklch(0.55 0.2 260)
                 accent-ring-red-500, accent-ring-[#0f0]
addVariant       hocus:underline  → :is(:hover, :focus-visible)
matchVariant     nth-2:font-bold  → :nth-child(2);   nth-odd:italic → :nth-child(odd)
addComponents    .pill  → appeared in the UTILITIES layer; "pill p-8" gave padding 32px
```

Three lessons from running it, each of which cost me a wrong first version:

- **Name collisions are silent.** My first plugin utility was called `text-shadow-md`. Tailwind already has `text-shadow-*` utilities, and the output was one rule with *both* declarations, the built-in's three-layer shadow and then mine, so mine won only because it came last. I renamed it `glow-text-*`. Check a name against the built-in list before you ship it.
- `theme()` **returns the literal value, not a** `var()`**.** `accent-ring-brand` compiled to `oklch(0.55 0.2 260)`, whereas the `@utility glow-*` from earlier compiled to `var(--color-brand)`. A plugin utility therefore does not follow a variable that you re-define per theme (chapter 2). Use `@utility` with `--value()` when theming matters.
- `addComponents` **did not use the components layer** in this version: `.pill` was emitted with the utilities. It was still overridden by `p-8` and `px-8` (padding read from the browser: 12px/4px alone, 32px left with `px-8`, 32px all round with `p-8`), because it was written

before them, but do not depend on a layer guarantee. For a real component class, write `@layer components { … }` in CSS.

Rule of thumb: **reach for `@utility` first.** Move to a JavaScript plugin only when you need options, loops over data, or distribution through npm.

Legacy Configs: `@config`

A project moving from v3 can keep its `tailwind.config.js` with `@config "./tailwind.config.js";`. I wrote a config that used six common v3 options and checked which still took effect:

OPTION IN THE JS CONFIG	RESULT	EVIDENCE
`theme.extend.colors`	works	`bg-brand` generated
`theme.extend.spacing`	works	`p-18` generated
`plugins: [function …]`	works	a plugin utility generated
`darkMode: 'class'`	works	`dark:bg-black` became `.dark\:bg-black:is(.dark *)`
`content: [...]`	works (I did not expect this)	with `source(none)` and no `@source`, the classes in the listed file were still found; remove the line and nothing was scanned
`safelist: [...]`	**ignored**	`bg-red-500` in the list was not generated. Use `@source inline()` (chapter 1)
`corePlugins: { float: false }`	**ignored**	`float-left` was still generated

So a bridge config will keep most of a v3 theme working, but `safelist` and `corePlugins` fail silently, the same way as every other mistake in this course: no error, no rule. Chapter 8 walks through moving a v3 project fully into CSS.

MISTAKES WHILE PREPARING THIS CHAPTER

1. A broken `--value()` that did not fail the build. Covered above. **How to avoid it:** after every build that matters, check for the thing you expect, not just the exit code.

2. A plugin utility named like a built-in. `text-shadow-md` already exists. **How to avoid it:** look at

the generated rule, not just whether it "worked"; two declarations in one rule is the tell.

3. I repeated a name from my own outline. The outline said "compound @variant (v4.3)". The official page does not use that term and I could not date the feature, so the text above describes what I tested. **How to avoid it:** treat a planning note as a hypothesis until a run confirms it.

4. A command chain stopped quietly. I chained steps with `&&` after a helper whose last command was `grep`; `grep` exits 1 when it finds nothing, so the rest was skipped and my output looked empty. **How to avoid it:** use `;` when an earlier step may legitimately "fail", and read empty output as a question, not a result.

Hands-On Exercises

EXERCISE 1

Write functional utilities `tab-*`, `stack-*`, `glow-*` and `pad-x-*` (with a modifier), plus a negative `-nudge-*`. Use a class list that mixes valid and invalid values and list exactly which classes Tailwind generates. Then break one utility on purpose with a literal inside `--value()` and report the exit code and what was still generated.

View solution

EXERCISE 2

Define `theme-midnight`, `hocus`, `any-hover`, `desktop-hover` and a `dark-hover` built from `@variant dark` and `@variant hover`. Use stacked forms such as `md:theme-midnight:p-2` and `group-hocus:`, list the generated selectors, and then use `@variant` (with stacking and a comma list) inside a `.btn` rule of your own.

View solution

EXERCISE 3

Write a JavaScript plugin with options that adds a base rule, a static utility, a value-taking utility, a colour utility fed from `theme('colors')`, a variant and a `matchVariant`. Pass an option from the CSS, list what it generated, and read the computed padding of a plugin-defined class combined with `p-8` in Chrome.

View solution

CHAPTER 3 QUICK REFERENCE

- Choose: arbitrary value (one-off) → `@utility` (reusable class) → `@custom-variant` (new condition) → `@variant` (variants in your own CSS) → `@plugin` (packaged) → JS plugin (options, data, npm) → `@config` (bridge for old projects)
- `@utility name { … }` is static and works with all variants; a name that exists is **merged** with the built-in, yours last
- `@utility name-* { … --value(…) … }`: forms are theme key `--value(--x-*)`, bare type `integer`, arbitrary `[length]`, literal `"x"`; `--spacing()`, `--modifier()`, `--default()`; negatives need their own `-name-*`
- A class whose value does not match is **not generated**. A bad `--value()` prints an error but the build exits 0
- `@custom-variant name (&:selector);` or a block with `@slot`; wrap in `:where()` for zero specificity; nest at-rules for two conditions; compose built-ins with `@variant`
- `@variant hover { … }` inside your CSS; `hover:focus` stacks, `dark, print` repeats. Your own unlayered rules still beat utilities
- `@plugin "@tailwindcss/typography"` (`prose`, `className` renames, not aliases) and `@plugin "@tailwindcss/forms"` (`strategy: class` to avoid restyling bare inputs)
- JS plugin: `plugin.withOptions`; `addBase`, `addUtilities`, `matchUtilities`, `addVariant`, `matchVariant`; `theme()` gives literal values, not `var()`; `addComponents` landed in the utilities layer; avoid built-in names
- `@config`: `theme.extend`, `plugins`, `darkMode` and `content` work; `safelist` and `corePlugins` are ignored

COMING NEXT

Chapter 4 goes deep on the variants Tailwind already has: named groups and peers (`group/card`, `peer/field`), `has-*`, `not-*`, `in-*` and `nth-*`, the `*:` and `**:` child variants, `data-*` state machines, `starting:` and logical variants. Start with *Tailwind CSS Intermediate/Advanced 4*.

Tailwind CSS Intermediate/Advanced

Chapter 4 · Advanced Variants and Selectors

Tailwind CSS Fundamentals 6 covered hover, focus, groups and peers. A variant is a selector in disguise, and v4 turns most of the useful modern selectors (`:has()`, `:not()`, `:nth-child()`, `@starting-style`) into class prefixes. This chapter goes through the ones that change how you build interfaces, and for each one it shows the selector Tailwind generates and what a browser did with it, because several of them behave differently from what the name suggests.

> **EVERY EXAMPLE HERE WAS RUN**
>
> Tailwind CLI 4.3.3 and Chrome 154, driven by a script that sent real mouse movement and clicks and then read computed styles. Not tested: Firefox, Safari, touch devices (where it matters the text says so), and screen readers.

Named Groups and Peers

`group` lets a child react to an ancestor's state. The trap is nesting: `group-hover:` means "*any* ancestor with the class `group` is hovered". Naming the group (`group/card`) makes the link exact. I built a card containing an item, both groups, with three spans that react differently, and moved the pointer:

```
<div class="group/card group">                    <!-- the card -->
  <div class="h-8">card only</div>
  <div class="group/item group">                  <!-- the item inside it -->
    <span class="group-hover/card:text-red-500">card</span>
    <span class="group-hover/item:underline">item</span>
    <span class="group-hover:italic">plain</span>
  </div>
</div>
```

```
                    group-hover/card: colour   group-hover/item: underline   group-hov
nothing hovered     normal                     none                          normal
hover card only     RED                        none                          italic
hover the inner item RED                       underline                     italic

.group-hover\/card\:text-red-500:is(:where(.group\/card):hover *)
```

```
.group-hover\/item\:underline:is(:where(.group\/item):hover *)
.group-hover\:italic:is(:where(.group):hover *)          (any ancestor named plain "grou
```

Peers work sideways: a sibling reacts to the state of an earlier sibling. They compile to the general-sibling combinator, which only looks *forward* in the markup. Measured with a text input marked `peer/field` and a paragraph on each side of it:

```
<p class="peer-focus/field:text-red-500">before</p>
<input class="peer/field border">
<p class="peer-focus/field:text-red-500">after</p>
```

```
before focus:   before normal   after normal
field focused:  before normal   after RED

.peer-focus\/field\:text-red-500:is(:where(.peer\/field):focus ~ *)
```

The paragraph before the input never reacts: `~` cannot look backwards. If a label must appear above the input, put the input first in the HTML and reorder visually (flex `order` or a grid). The same `/name` suffix applies to `group-has-*`, `group-data-*` and `peer-*` variants.

`has-*`: A Parent That Reacts to Its Children

`has-*` wraps CSS `:has()`, which finally lets a parent change based on what is inside it. The shorthand variants take the state name; the bracket form takes any selector:

```
<label class="has-checked:bg-green-500 has-[input:disabled]:opacity-50"><input type="che

.has-checked\:bg-green-500:has(:checked)
.has-\[input\:disabled\]\:opacity-50:has(:is(input:disabled))
.has-\[\>img\]\:p-2:has( > img)                          (direct child only)
.group-has-\[img\]\/card\:p-4:is(:where(.group\/card):has(:is(img)) *)
```

```
unchecked label bg:                  oklch(0.967 0.003 264.542)   (gray-100)
disabled-checkbox label opacity:     0.5
after clicking the checkbox, label bg: green-500
```

This replaces a lot of JavaScript: a selected radio card, a form row that highlights when a field inside it is invalid (`has-[:invalid]`), a card that changes layout when it contains an image (`has-[img]`). Use the bracket form with a simple selector; a long selector inside `has-[…]` is hard to read and a sign that a data attribute (below) would be clearer.

`not-*`: Negation

`not-` inverts another variant or accepts a selector. The generated selectors show what each one really means:

```
not-first:mt-2              .…:not(:first-child)
not-[.skip]:underline       .…:not(:is(.skip))
not-has-checked:bg-red-500  .…:not(:has(:checked))
not-md:hidden               @media not (width >= 48rem)        (below the md breakpoint)
not-hover:underline         .…:not(:hover)   AND   @media not (hover: hover) { … }
not-supports-[display:grid]:block   @supports not (display:grid) { … }
```

```
<li class="not-first:bg-blue-500 not-[.skip]:underline">1</li>      bg none      underli
<li class="not-first:bg-blue-500 not-[.skip]:underline skip">2</li>  bg blue-500  underl
```

Notice `not-hover:`: it generates *two* rules, the plain `:not(:hover)` and a media query for devices that cannot hover at all. By reading the CSS, the second one means a touch-only device receives the `not-hover` styling permanently. I did not test that on a phone. If you use it, think about what that device should show.

`in-*`: A Group Without the Class

`in-*` is `group-*` without naming an ancestor: "*some* ancestor matches this". It is handy for state set by a library on an element you do not control (`in-data-[state=open]:flex`, `in-[.dark]:text-white`). But there is one very easy trap:

```
<div class="h-8 bg-gray-100"><span class="in-hover:text-red-500">in</span></div>

.in-hover\:text-red-500 compiles to   :where(:hover) .in-hover\:text-red-500
```

```
pointer never moved:                  span normal   (body:hover false)
pointer over an unrelated label:      span RED      (body:hover true, the span's own pa
```

"Some ancestor is hovered" includes `body` and `html`, so as soon as the pointer is anywhere over the page the span turns red, even though the pointer is nowhere near its parent. The same applies to `in-focus:` when focus is inside the page. Use `in-*` only with a specific ancestor (`in-data-[…]`, `in-[.class]`), and use a named `group` for hover and focus.

`nth-*`: Position

Tailwind has `first`, `last`, `odd`, `even`, and the general `nth-*` family, which takes a number, a formula in brackets, or the `last` and `of-type` forms. I put these four on every item of a five-item list:

```
<li class="nth-3:bg-red-500 nth-[2n+1]:underline nth-last-2:italic odd:font-bold">

nth-3          :nth-child(3)          nth-last-2    :nth-last-child(2)
nth-[2n+1]     :nth-child(2n+1)       odd           :nth-child(odd)         nth-of-type-2
```

```
li1   underline  bold
li2
li3   bg red-500  underline  bold
li4   italic
li5   underline  bold
```

`nth-[2n+1]` and `odd` match the same items (1, 3, 5); `nth-last-2` counts from the end, so it is item 4 of 5. Prefer these over adding classes in a loop when the pattern is purely positional (striped rows, a different first item).

`*:` and `**:`: Styling Children From the Parent

`*:` applies a utility to every direct child and `**:` to every descendant. They are tempting for lists of plain items ("every child gets padding") and they have one sharp edge, which I measured with a parent containing a plain child and a child with its own classes:

```
<div class="*:p-2 *:bg-yellow-500 **:text-red-500">
  <div>child<span>grandchild</span></div>
  <div class="p-8 bg-blue-500">own classes</div>
</div>

:is(.\*\:p-2 > *)           and      :is(.\*\*\:text-red-500 *)
```

```
plain child:       padding 8px   bg yellow-500   grandchild colour RED
child with p-8 bg-blue-500:   padding 8px   bg yellow-500   colour RED
```

The second line is the lesson: **the child's own `p-8` and `bg-blue-500` lost.** The parent's rule has the same specificity (one class) as the child's utility and is written later in the stylesheet (in my build `.p-8` was at line 188 and the `*:p-2` rule at line 210), so it wins. The practical rules: use `*:` only for children that will never carry their own styling; do not mix it with per-child classes; and treat `**:` as a last resort, because it reaches every descendant including ones you add later and makes the markup hard to read (a reader looking at a child cannot see where its colour comes from).

Element State: `open`, `inert` and Popovers

Native elements keep their own state, and Tailwind has variants for it. These are the selectors:

```
open:bg-yellow-500      .…:is([open], :popover-open, :open)
inert:opacity-50        .…:is([inert], [inert] *)
```

```
details with open:bg-yellow-500, closed             bg transparent
details after det.open = true                       bg yellow-500
popover with open:bg-green-500, hidden / shown      white / green-500
<div inert class="inert:opacity-50"> opacity        0.5     (and a child with inert:und

popover-open:font-bold   weight 400 shown  -- no rule was generated for popover-open: at
```

Three notes. `open:` in v4 covers `[open]` (details, and dialogs, which I did not test), a shown popover (tested) and the `:open` pseudo-class in one variant, so use it instead of `popover-open:`: I wrote `popover-open:font-bold` and Tailwind generated nothing for it (no error). `inert:` matches the element *and everything inside it* (`[inert] *`), which is what you want for

dimming a section while a modal is open. And a test-harness caveat: my scripted click on the `<summary>` did not toggle the `details` element, so I set `det.open = true` from script instead; the variant responded to the attribute, but I cannot confirm the click-to-toggle path in that harness.

`data-*` and `aria-*`: State Machines in Attributes

The cleanest way to style a component with several states is not a pile of conditional classes but one attribute that holds the state, with classes that react to its values. Headless UI libraries work this way (they set `data-state` or `aria-*` attributes and leave the styling to you; chapter 6). The three forms:

VARIANT	SELECTOR	USE FOR
`data-[state=active]:bg-blue-500`	`[data-state="active"]`	an attribute with a specific value
`data-active:underline`	`[data-active]`	an attribute that is just present
`aria-selected:font-bold`	`[aria-selected="true"]`	accessibility state that is already on the element
`group-data-[state=open]/menu:rotate-180`	`…:is(:where(.group\/menu)[data-state="open"] *)`	a child reacting to a named ancestor's state

```
<button data-state="active"   aria-selected="true"  class="data-[state=active]:bg-blue-5
<button data-state="inactive" aria-selected="false" class="data-[state=active]:bg-blue-5
```

```
as loaded:        tab1  bg blue-500  weight 700  underline none      tab2  bg none  weig
after JavaScript sets data-state, aria-selected and data-active on tab2 (and resets tab1
                  tab1  bg none  weight 400                         tab2  bg blue-500
menu chevron rotate:  closed none  ->  data-state="open" on the group: 180deg
```

No class was added or removed: the script changed three attributes and the styles followed. `aria-selected:` is doubly useful, because the same attribute is what assistive technology reads, so the styling and the accessibility state cannot drift apart. Note that `rotate-180` sets the CSS

`rotate` property in v4, not `transform`: reading `transform` in the browser returned `none` in both states, which fooled me for a moment.

`starting:` : Entry Transitions Without JavaScript

A newly inserted element has no "before" state, so it cannot transition in. The CSS rule `@starting-style` supplies one, and `starting:` is its variant:

```
<div class="transition-opacity duration-500 opacity-100 starting:opacity-0">entering</di

@starting-style { .starting\:opacity-0 { opacity: 0; } }
```

```
opacity just after insertion: 0     two animation frames later: 0.00247     after 800 ms
```

The element appeared at 0, was already fading in two frames later, and settled at 1, with no timer or class toggling. I tested an element inserted by script; dialogs and popovers use the same mechanism but I did not run them. It worked in Chrome 154; I did not test other browsers, so check your support target.

Direction: `rtl:` and `ltr:`

Two variants key on the text direction, and they pair with the logical utilities (`ms-*`, `me-*`, `ps-*`) that flip on their own:

```
<div dir="rtl"><p class="ms-4 ltr:text-left rtl:text-right">…</p></div>

.rtl\:text-right:where(:dir(rtl), [dir="rtl"], [dir="rtl"] *)
```

```
dir="rtl":  ms-4 -> margin-left 0px, margin-right 16px     text-align right
dir="ltr":  ms-4 -> margin-left 16px, margin-right 0px     text-align left
```

`ms-4` (margin at the start) landed on the right in a right-to-left block without an `rtl:` class, which is why logical utilities are preferred and `rtl:` is for what has no logical form. The full treatment is in chapter 9.

MISTAKES WHILE PREPARING THIS CHAPTER

1. My filter hid half of the output. I listed generated selectors with a pattern that kept lines starting with a dot, and concluded that `*:`, `**:` and `in-hover:` were not generated. They were: their selectors start with `:is(` or `:where(`. (`popover-open:` really was missing.) **How to avoid it:** when something seems absent, search the whole file for the class name before deciding.

2. A failed command left the page unedited. A quoting error stopped the shell before my page edit ran, and the next step failed with a confusing browser error. **How to avoid it:** after any failed command, check that its side effects happened before running the next step.

3. My first in-hover test measured the wrong thing. I read it after earlier clicks had left the pointer on the page, so a result labelled "pointer away" was red. That led me to the real behaviour (it is red whenever the pointer is on the page), and I rebuilt the test to start with a pointer that had never moved. **How to avoid it:** print the precondition (`body:hover`) next to the result.

4. I read the wrong property. `rotate-180` sets `rotate`, not `transform`. **How to avoid it:** read the generated rule first, then choose what to measure.

Hands-On Exercises

EXERCISE 1

Build a card containing an item, both marked as named groups, with one span for each of `group-hover/card:`, `group-hover/item:` and the unnamed `group-hover:`. Move the pointer over nothing, the card only and the item, and record each span. Then add a paragraph before and after a `peer/field` input and show which one reacts when the input is focused.

View solution

EXERCISE 2

Write a checkbox label with `has-checked:` and a disabled one with `has-[input:disabled]:`; a list using `not-first:` and `not-[.skip]:`; a five-item list with `nth-3:`, `nth-[2n+1]:`, `nth-last-2:` and `odd:`; and an `in-hover:` span. Record all results, and show with `body:hover` why `in-hover:` is not safe.

View solution

EXERCISE 3

Use `*:p-2`, `*:bg-yellow-500` and `**:text-red-500` on a parent with one plain child and one child that has its own `p-8 bg-blue-500`, and find out who wins and why. Then build a two-tab strip driven by `data-state`, `aria-selected` and `data-active`, a chevron rotated by `group-data-[state=open]/menu:`, and an element that fades in with `starting:`.

View solution

CHAPTER 4 QUICK REFERENCE

- Name groups and peers (`group/card`, `group-hover/card:`, `peer/field`, `peer-focus/field:`); unnamed `group-hover:` matches *any* ancestor named group. Peers react only to **earlier** siblings
- `has-checked:`, `has-[input:disabled]:`, `has-[>img]:`, `group-has-[img]/card:`: a parent reacting to its children (`:has()`)
- `not-first:`, `not-[.skip]:`, `not-has-checked:`, `not-md:` (below md), `not-supports-[…]:`; `not-hover:` also adds a no-hover-device media rule
- `in-*` = some ancestor matches; `in-hover:` and `in-focus:` match as soon as the pointer or focus is anywhere on the page. Use `in-data-[…]` or `in-[.class]`
- `nth-3:`, `nth-[2n+1]:`, `nth-last-2:`, `nth-of-type-2:`, `odd:`, `even:`, `first:`, `last:`
- `*:` (children) and `**:` (descendants) beat the child's own utilities of equal specificity; avoid mixing them; `**:` is a last resort
- `open:` = `[open]`, `:popover-open` or `:open`; no `popover-open:` rule in 4.3.3; `inert:` also matches everything inside
- `data-[state=x]:` (value), `data-active:` (present), `aria-selected:`, `group-data-[state=open]/menu:`: state lives in an attribute, styles follow
- `starting:` = `@starting-style`: entry transitions for inserted elements (Chrome 154 tested)
- `rtl:` / `ltr:` check `dir`; prefer logical utilities (`ms-4`)

COMING NEXT

Chapter 5 takes everything so far into real projects: Vite, Astro, Next.js, SvelteKit, React Router and Vue, no-flash dark mode with server rendering, framework style blocks and `@reference`, and monorepo theme packages. Start with *Tailwind CSS Intermediate/Advanced 5*.

Tailwind CSS Intermediate/Advanced

Chapter 5 · Tailwind in Frameworks

So far every build used the Tailwind command line. Real projects run Tailwind inside a bundler or a framework, and the integration has the same three jobs everywhere: read your CSS entry file, scan the project for classes, and put the result in the page. This chapter wires Tailwind into seven toolchains, runs each one for real, and then deals with the problems that appear once it is in: style blocks in components, a flash of the wrong theme when the server renders the page, and shared code in a monorepo.

> **EVERY EXAMPLE HERE WAS RUN**
>
> Tailwind 4.3.3 everywhere, on Node 24.18.0 and Windows 11. Versions installed: Vite 8.3.4, Astro 7.3.8, Next.js 16.4.0 (Turbopack), SvelteKit 3.0.1 with `@sveltejs/adapter-static` 4.0.0 and Svelte 5.57.2, React Router 8.4.0, Vue 3.5.43 with `@vitejs/plugin-vue` 6.0.10, webpack 5.111.1. I ran production builds for all of them and the development server for Vite and Next.js. **Not run:** Nuxt, Remix and other frameworks, Rspack, Parcel, the other development servers, real deployments, and any browser except Chrome 154. Where I name something I did not run, it says so.

Which Package for Which Tool

YOUR TOOL	INSTALL	CONFIGURATION I RAN
Vite, and anything built on it (Astro, SvelteKit, React Router, Vue)	`@tailwindcss/vite`	`plugins: [tailwindcss()]` in the Vite config (Astro: inside the `vite` key)
Next.js, and other PostCSS-based tools	`@tailwindcss/postcss`	`plugins: { "@tailwindcss/postcss": {} }` in `postcss.config.mjs`
webpack	`@tailwindcss/webpack`	a **loader** after `css-loader` (below)
Nothing (static HTML, server templates)	`@tailwindcss/cli`	chapter 1 of this course

In every case the CSS entry file is the same one line, `@import "tailwindcss";`, plus your `@theme`. In every case I built a small page with a few text utilities and checked that the built CSS

contained the rules. Each of the seven worked on the first correct configuration; the surprises are below.

Vite

```
// vite.config.js
import { defineConfig } from 'vite';
import tailwindcss from '@tailwindcss/vite';
export default defineConfig({ plugins: [tailwindcss()] });

/* src/style.css */
@import "tailwindcss";
```

```
vite build   exit 0     dist/assets/index-CtRtC2ic.css  4.85 kB
  text-3xl, font-bold, underline, text-sky-600: each present in the file; the file is al

vite dev (port 5199): GET /src/style.css  -> 6,118 bytes of CSS-as-a-module, contains te
  edit index.html to add bg-red-500, request again (no restart):  bg-red-500 now present
```

Two lessons. A production build minifies the CSS without being asked (the CLI needed `--minify`, chapter 1). And the development server scans your HTML as you edit it, so a new class appears without restarting anything. I stopped the server by finding the process listening on the port and ending only that one.

Astro, Next.js, SvelteKit and React Router

These four use the same two packages in four different places. All four built without errors:

FRAMEWORK	WHERE THE ONE LINE GOES	BUILT CSS (MY TEST PAGE)
Astro 7.3.8	`astro.config.mjs`: `vite: { plugins: [tailwindcss()] }`; the page imports `../styles/global.css` in its frontmatter	6,295 B, includes the page's own `<style>` block
Next.js 16.4.0	`postcss.config.mjs` as above; `app/layout.js` imports `./globals.css`	4,855 B, linked as `/_next/static/chunks/….css`

FRAMEWORK	WHERE THE ONE LINE GOES	BUILT CSS (MY TEST PAGE)
SvelteKit 3.0.1	`vite.config.js`: `plugins: [tailwindcss(), sveltekit({ adapter: adapter() })]`; `+layout.svelte` imports `../app.css`	4,956 B global, plus a 264 B file for the component's style block
React Router 8.4.0 (SPA mode)	`vite.config.ts`: `plugins: [tailwindcss(), reactRouter()]`; `root.tsx` imports `./app.css`	4,802 B, linked from `build/client/index.html`

(The sizes are for different pages, so do not compare them with each other; they only show that real CSS came out.) Three things I found by running them rather than reading about them:

- **SvelteKit 3 no longer reads `svelte.config.js`.** My first attempt used the file I remembered and the build stopped with `SvelteKit error: config_file_unsupported`: the message says to pass the configuration to the `sveltekit(...)` plugin in the Vite config, which is what the table shows. Order in the plugin list: I put `tailwindcss()` before `sveltekit()`; I did not test the reverse.
- **Next.js wrote the theme colour twice.** The same `--color-brand: oklch(0.55 0.2 260)` came out as `oklch(55% .2 260)` in Vite, Astro and SvelteKit, but in the Next.js build as `#1c69e3` followed by `@supports (color:lab(0% 0 0)) { :root,:host { --color-brand: lab(45.7354% 14.1968 -69.775) } }`. The two tools use different browser targets when they post-process the CSS. I did not trace the setting that causes it; if exact colour matters across tools, compare their built output.
- **Next.js warned about my folder layout.** It printed `We detected multiple lockfiles and selected the directory of C:\lwdj\tw2\package-lock.json as the root directory`, because my scratch project sat inside another one. That is a monorepo question in disguise; see the last section.

webpack

The webpack integration is a **loader**, not a plugin. It goes last in the `use` list (loaders run right to left), after `css-loader` and the extraction plugin:

```
module: { rules: [{
  test: /\.css$/,
  use: [MiniCssExtractPlugin.loader, 'css-loader',
```

```
        { loader: '@tailwindcss/webpack', options: { optimize: { minify: true } } }],
}] },
```

```
without options:                        main.css 5,362 bytes   (readable, 191 lines)
optimize: { minify: true }:             main.css 4,770 bytes
the options in the type definitions:    base (directory to scan, default the working direc
```

The loader scans from the working directory, so run webpack from the project root (or set `base`). The page's classes were found in `src/index.js` without any extra configuration.

Component Styles: `@apply` and `@reference`

The most common v4 integration error comes from the framework's own style blocks (Vue, Svelte, Astro) and CSS modules. Each block is compiled as its own stylesheet, so it cannot see your theme or utilities. This is what happened in a Vue single-file component:

```
<style scoped>
.btn { @apply bg-brand text-white px-4 py-2; }
</style>
```

```
exit 1
Error: Cannot apply unknown utility class `bg-brand`. Are you using CSS modules or simil
```

The fix is `@reference`, which imports your main stylesheet for use by `@apply` without copying any of it into the output. I measured the alternatives with the same component:

VERSION OF THE COMPONENT	RESULT	OUTPUT CSS
`@apply`, nothing else	build fails	–
+ `@reference "./style.css";`	works	4,755 B
`@import "tailwindcss";` inside the component	works, but repeats the whole reset (box-sizing rule 4 times instead of 2)	10,054 B
no `@apply`: `background: var(--color-brand)` and friends	works with no `@reference` at all	4,463 B

VERSION OF THE COMPONENT	RESULT	OUTPUT CSS
no style block: utilities in the template	works	4,601 B

The same error and the same fix applied to a CSS module in plain Vite (`._btn_16twv_2{…}` came out). Astro and SvelteKit style blocks with `@reference "../styles/global.css"` built and produced `.btn[data-astro-cid-…]` and `.btn.svelte-…` rules; the relative path is from the component, and my first attempt with the wrong depth failed with `Can't resolve`.

WHAT TO DO

Tailwind's own Compatibility page recommends avoiding `<style>` blocks in Vue, Svelte and Astro components, because each block is processed separately (so Tailwind runs once per block), and prefers plain CSS variables over `@apply`. The sizes above agree. **Use utility classes in the template** and keep component style blocks for what utilities cannot do. Theme values are ordinary CSS variables (`var(--color-brand)`, `calc(var(--spacing) * 4)`), and those work in any style block with no setup. Reach for `@apply` plus `@reference` only when you must, and never import all of Tailwind into each component.

Server Rendering and the Flash of the Wrong Theme

Chapter 2 switched themes with a `data-mode` attribute. If the server renders the page and your application code sets the attribute later, a visitor who chose dark sees a light page first. I measured it with three pages that differ only in *when* the attribute is set, saved preference "dark", sampling the background on every animation frame:

```
flash.html     late script (150 ms), standing in for a framework applying it after it lo
noflash.html   classic inline <script> in <head>, before the stylesheet
module.html    the same script with type="module"
```

```
flash.html:    0 ms  oklch(0.99 0 0)   light         175 ms  oklch(0.2 0.02 260)   dark
noflash.html:  0 ms  oklch(0.2 0.02 260)   dark from the first sample
module.html:   0 ms  oklch(0.99 0 0)   light         1 ms    oklch(0.2 0.02 260)   dark
```

The answer is a small **classic** script in the document head that sets the attribute before the page paints. The module version was deferred by the browser even though it was inline, so its first

frame was light. How that looks in two frameworks I built:

```
--- Astro ---
<script>document.documentElement.dataset.mode = localStorage.getItem('mode') || 'light';
<script is:inline>document.documentElement.dataset.mode = localStorage.getItem('mode') |

--- Next.js (app/layout.js) ---
<html lang="en" suppressHydrationWarning>
  <head><script dangerouslySetInnerHTML={{ __html: init }} /></head>
```

```
Astro build, <head> of dist/index.html:
  <script type="module">document.documentElement.dataset.mode=localStorage.getItem(`mode
  <script>document.documentElement.dataset.mode = localStorage.getItem('mode') || 'light
  <link rel="stylesheet" href="/_astro/index.DAT-mcAi.css">

Next.js dev server, saved mode dark, console after load:
  script in <head>, no suppressHydrationWarning:  error: A tree hydrated but some attribu
  with suppressHydrationWarning on <html>:        no error;  data-mode on <html> = dark
```

Astro rewrote my plain `<script>` into `<script type="module">`, which is the deferred kind, so use `is:inline` to keep it a classic script. In Next.js, a script that changes `<html>` before React hydrates makes the server HTML and the browser's disagree, and React says so; `suppressHydrationWarning` on that one element is the intended answer and it removed the message. I did not run SvelteKit (the equivalent place is the inline script in `src/app.html`) or React Router (a script in the root layout) for this, so treat those two as untested.

Monorepos and Shared Theme Packages

A monorepo splits code into packages: here a `theme` package (tokens), a `ui` package (components that carry Tailwind classes) and an app that uses both through npm workspaces. Two things break silently, and I measured both with the same built app:

```
packages/theme/theme.css:  @theme { --color-brand: …; --color-accent: …; }
packages/ui/index.js:      export const button = 'rounded-lg bg-brand px-4 py-2 text-whi

apps/web/src/style.css (three versions):
  1:  @import "tailwindcss";   @import "@acme/theme/theme.css";
```

```
  2:  …as 1, plus   @source "../../../packages/ui";
  3:  @import "tailwindcss";   @source "../../../packages/ui";   (theme import forgotten
```

	1 (no @source)	2 (theme + @source)	3 (@source, no theme)
bg-brand	0	1	0
text-white	0	1	1
px-4	0	1	1
hover:bg-accent	-	1	-
text-accent (app's own)	1	1	0
--color-brand variable	absent	oklch(55% .2 260)	absent

Version 1: the shared button's classes live outside the app's folder and the scanner does not follow the workspace link in `node_modules`, so none of them were generated. Version 3: without the theme import the tokens do not exist, so even the app's own `text-accent` disappeared. Neither raised an error. The rules: add `@source` for every workspace package whose files contain classes (the path is relative to the CSS file), and import the shared theme CSS in every app, once. The theme package is just a CSS file exposed through the package's `exports`, which makes it a convenient place to put the semantic tokens from chapter 2 for several apps.

MISTAKES WHILE PREPARING THIS CHAPTER

1. I wrote SvelteKit 3 configuration from memory. The old `svelte.config.js` approach is rejected; the error message told me the new place. **How to avoid it:** run the version you are teaching before writing its configuration.

2. A wrong relative path in `@reference`. I wrote `../style.css` from a file that was already beside `style.css`. **How to avoid it:** read the "Can't resolve" line; it names the folder it searched.

3. My count of a class gave a false zero. The pattern `hover.bg-accent` could not match `hover\:bg-accent` (two characters between). **How to avoid it:** when a count is zero where you expect one, print the matching line from another angle before concluding it is missing.

4. I nearly compared CSS sizes across different pages. The framework builds contain different markup, so their sizes mean nothing side by side; only the component-style comparison (same component, five versions) is a fair one. **How to avoid it:** change one thing at a time.

Hands-On Exercises

EXERCISE 1

Make three pages that apply a saved dark preference by a late script, a classic inline script in the head, and an inline module script. Sample the page background on every animation frame for 400 ms with the preference set to dark and report when each page turns dark. Then say which of the three your framework's layout should use.

View solution

EXERCISE 2

In a Vue (or Svelte or Astro) project, put `@apply bg-brand text-white px-4 py-2` in a component style block. Show the error, fix it with `@reference`, then measure the CSS for: the fix, importing `tailwindcss` inside the component, plain `var()` theme values with no `@apply`, and utilities in the template.

View solution

EXERCISE 3

Build an npm-workspaces monorepo with a theme package, a UI package and an app. Produce the three results table: no `@source`, theme plus `@source`, and `@source` without the theme import. Explain in each case what is missing and why nothing complained.

View solution

CHAPTER 5 QUICK REFERENCE

- Vite-based tools (Vite, Astro, SvelteKit, React Router, Vue): `@tailwindcss/vite`, `plugins: [tailwindcss()]`. Next.js / PostCSS: `@tailwindcss/postcss`. webpack: the `@tailwindcss/webpack` **loader** (options `base`, `optimize`). No bundler: the CLI
- CSS entry in every case: `@import "tailwindcss";` plus your `@theme`; import it once from the app's root
- Vite builds minify automatically; the dev server picks up new classes without a restart
- SvelteKit 3: configuration goes into `sveltekit({ … })` in the Vite config, not `svelte.config.js`
- Component style blocks and CSS modules cannot see Tailwind: `@reference "./style.css";` (path relative to the component) before `@apply`; never `@import "tailwindcss"` in each component (10,054

B against 4,755 B)

- Theme values are plain variables: `var(--color-brand)` works in any style block without `@reference`
- No-flash theme: a **classic** inline script in the head sets `data-mode` before paint. Astro: `is:inline`. Next.js: `dangerouslySetInnerHTML` in `<head>` and `suppressHydrationWarning` on `<html>`. Module scripts are deferred
- Monorepo: `@source "../../../packages/ui"` for each package with classes (relative to the CSS file) and `@import "@acme/theme/theme.css"` in each app. Both failures are silent
- Build tools post-process the CSS differently: the same token came out as `oklch()` in Vite and as `#hex` plus a `lab()` override in Next.js

COMING NEXT

Chapter 6 turns to component patterns: `cva` for typed variants, `clsx` and `tailwind-merge`, headless libraries (Radix, Headless UI, React Aria) styled with Tailwind, shadcn/ui, daisyUI and Flowbite compared, and when a plain component beats all of it. Start with *Tailwind CSS Intermediate/Advanced 6.*

Tailwind CSS Intermediate/Advanced

Chapter 6 · Component Patterns and Libraries

Utility classes are fine in a page. Once you wrap them in a component (a button with a `variant` and a `size`, which callers can also give an extra `className`) three problems appear: which of two conflicting classes wins, how to describe the variants without a pile of `if` statements, and where the behaviour (open, closed, selected, focus) comes from. This chapter takes each problem with the tool people actually use, runs it, and then compares the ready-made libraries. *CSS Frameworks* 7 covers shadcn/ui as an idea; here I install it and read what it wrote.

> **EVERY EXAMPLE HERE WAS RUN**
>
> Tailwind 4.3.3, Node 24.18.0, Windows 11, headless Chrome 154 with real mouse clicks. Packages: react and react-dom 19.3.0, vite 8.3.4, clsx 2.1.1, tailwind-merge 3.7.0, class-variance-authority 0.7.1, @radix-ui/react-tabs 1.2.0 and react-dialog 1.3.0, @headlessui/react 2.2.10, react-aria-components 1.22.1, daisyui 5.7.47, flowbite 4.0.2, shadcn 4.21.4 (CLI), TypeScript 7.0.2. **Not run:** Vue or Svelte versions of these libraries, keyboard-only testing and screen readers, React Aria's own Tailwind plugin, Flowbite's framework packages, other shadcn styles, and any browser except Chrome 154.

Problem 1: Which Class Wins?

A component sets `bg-blue-500` and a caller passes `bg-red-500`. Both end up on the element. You might expect the later one in the attribute to win. It does not, and the reason is the CSS file, not the HTML. I built both classes and rendered four elements in Chrome:

```
<div id="a" class="bg-red-500 bg-blue-500">a</div>
<div id="b" class="bg-blue-500 bg-red-500">b</div>
<div id="c" class="px-8 px-2">c</div>
<div id="d" class="px-2 px-8">d</div>
```

```
built CSS order:  .bg-blue-500 {...}  .bg-red-500 {...}  .px-2 {...}  .px-8 {...}

a  class="bg-red-500 bg-blue-500"   background oklch(0.637 0.237 25.331)   red
b  class="bg-blue-500 bg-red-500"   background oklch(0.637 0.237 25.331)   red     same
c  class="px-8 px-2"                padding-left 32px                       px-8
d  class="px-2 px-8"                padding-left 32px                       px-8
```

Tailwind writes the rules in its own order (red after blue, `px-8` after `px-2`) and with equal specificity the later rule in the file wins. So the attribute order is irrelevant. A caller who adds `bg-blue-500` to a component that already has `bg-red-500` may get no change at all, depending on a sort order they cannot see. The fix is to remove the loser from the string before it is rendered. That is what `tailwind-merge` does.

clsx and tailwind-merge

Two small libraries, two jobs. `clsx` joins class values and drops the false ones. `twMerge` from `tailwind-merge` knows which Tailwind classes conflict and keeps the last of each conflicting group. The usual helper is named `cn` and combines them:

```
import { clsx } from 'clsx';
import { twMerge } from 'tailwind-merge';

export const cn = (...inputs) => twMerge(clsx(inputs));
```

```
clsx("px-2", false, "px-8")                    "px-2 px-8"          clsx only joins; not
twMerge("px-2 px-8")                           "px-8"
twMerge("p-4 px-2")                            "p-4 px-2"           px-2 is more specifi
twMerge("px-2 p-4")                            "p-4"                p-4 covers px-2, so
twMerge("bg-red-500 bg-blue-500")              "bg-blue-500"
twMerge("rounded-md rounded-lg")               "rounded-lg"
twMerge("w-4 size-8")                          "size-8"
twMerge("hover:bg-red-500 bg-blue-500")        "hover:bg-red-500 bg-blue-500"    differe
```

The last line is a trap, not a bug. A component with `bg-blue-600 hover:bg-blue-700` and a caller who passes `bg-green-600` gets `bg-green-600` but keeps the component's `hover:bg-blue-700`, so the button turns blue when hovered. I confirmed this with the helper above (the result is in the cva section below). The caller has to override the hover class as well.

Custom theme names confuse the merge

`twMerge` does not read your `@theme`. It recognises the built-in names. I added `--text-huge: 4rem` and `--color-brand` to a theme (chapter 2) and merged:

```
twMerge("text-huge text-lg")           // two font sizes
twMerge("text-brand text-red-500")     // two colours, one custom
```

```
const tm = extendTailwindMerge({
  extend: { classGroups: { 'font-size': [{ text: ['huge'] }] } },
});
tm("text-lg text-huge")
tm("text-huge text-lg")
```

```
twMerge("text-huge text-lg")        "text-huge text-lg"      both kept: wrong, two sizes
twMerge("text-brand text-red-500")  "text-red-500"           right, an unknown name afte
tm("text-lg text-huge")             "text-huge"
tm("text-huge text-lg")             "text-lg"
```

Colours happen to work because an unknown `text-name` is treated as a colour. A custom font size does not, so the library has to be told with `extendTailwindMerge`. Any custom token in a size, shadow or radius group needs the same registration, or conflicts silently survive. (I only tested the two cases above; the library's own documentation describes the limitation.)

Problem 2: Variants Without a Mess

`cva` (class-variance-authority) turns a table of variants into a function that returns a class string. I wrote a button and called it:

```
import { cva } from 'class-variance-authority';

const button = cva('inline-flex items-center rounded-md font-medium', {
  variants: {
    intent: { primary: 'bg-blue-600 text-white hover:bg-blue-700', danger: 'bg-red-600 t
    size:   { sm: 'px-2 py-1 text-sm', md: 'px-4 py-2 text-base' },
  },
  compoundVariants: [{ intent: 'danger', size: 'sm', class: 'uppercase' }],
  defaultVariants: { intent: 'primary', size: 'md' },
});
```

```
button()
  inline-flex items-center rounded-md font-medium bg-blue-600 text-white hover:bg-blue-7
button({ intent: "danger", size: "sm" })
  inline-flex items-center rounded-md font-medium bg-red-600 text-white px-2 py-1 text-s
button({ size: "lg" })                      // a value that does not exist
  inline-flex items-center rounded-md font-medium bg-blue-600 text-white hover:bg-blue-7
```

```
button({ class: "px-8" })                    // the caller's extra class
  ... px-4 py-2 text-base px-8               both px-4 and px-8 kept: cva does not merge
twMerge(button({ class: "px-8" }))
  ... py-2 text-base px-8                    px-4 removed
cn(button(), "bg-green-600")
  inline-flex items-center rounded-md font-medium text-white hover:bg-blue-700 px-4 py-2
```

Four observations. The base string, the chosen variant classes and the compound variant (`uppercase` only for a small danger button) are joined in a fixed order, with defaults filling anything you did not pass. An unknown value (`size: "lg"`) silently contributes nothing; the button loses its size classes and nothing warns you at run time. `cva` only builds the string; resolving conflicts is a separate step, which is why components wrap it: `cn(button({ intent, size }), className)`. And the last line shows the hover trap from above: `bg-green-600` replaced `bg-blue-600` but `hover:bg-blue-700` is still there.

The warning for the unknown value comes from the types. I put this file in a TypeScript 7.0.2 project with strict mode and ran `tsc --noEmit`:

```
import { cva, type VariantProps } from "class-variance-authority";
const button = cva("btn", { variants: { size: { sm: "px-2", md: "px-4" } }, defaultVaria
type Props = VariantProps<typeof button>;
export const ok: Props = { size: "sm" };
export const bad: Props = { size: "lg" };
```

```
src/t.ts(5,29): error TS2322: Type '"lg"' is not assignable to type '"md" | "sm" | null
```

`VariantProps` gives the component's props their type for free, so a wrong variant is a compile error instead of a missing style. Without TypeScript you get the silent version.

The scanner still has to see whole class names

The class strings inside `cva` are ordinary strings in your source, so the scanner from chapter 1 finds them. The rule from that chapter still holds: write complete names (`'bg-red-600'`), never assemble them (`'bg-' + color + '-600'`). The variant maps in this chapter were built and their classes were present in the output.

Problem 3: Where Does the Behaviour Come From?

A tab list or a dialog needs state, keyboard handling, focus management and ARIA attributes. *Headless* libraries supply all of that and no styling. They expose their state as attributes on the elements, and Tailwind's attribute variants (chapter 4) style them. I built one page with three libraries and clicked them in Chrome, reading the attributes before and after:

```
// Radix: data-state="active" | "inactive"
<RTabs.Trigger value="a" className="data-[state=active]:bg-blue-500">One</RTabs.Trigger>

// Headless UI 2: boolean data attributes
<Tab className="data-selected:bg-blue-500 data-hover:underline data-focus:outline-2">One

// React Aria Components
<Tab className="data-selected:bg-blue-500 data-hovered:underline">One</Tab>
```

```
Radix tab 2, after clicking it:      aria-selected=true  data-state=active
Radix tab 1 (now inactive):          aria-selected=false data-state=inactive
Headless UI tab 2, after clicking:   aria-selected=true  data-headlessui-state="selected
Headless UI tab 1 (not selected):    aria-selected=false data-headlessui-state=""   (no
React Aria tab 2, after clicking:    aria-selected=true  data-hovered=true  data-selecte
React Aria tab 1 (not selected):     aria-selected=false (no data-selected)
Radix dialog trigger:                closed: aria-expanded=false data-state=closed   ope
Headless UI disclosure button:       closed: aria-expanded=false   open: aria-expanded=t
```

The three conventions differ and your classes must match the library:

LIBRARY	STATE ATTRIBUTES SEEN	TAILWIND CLASS THAT MATCHED
Radix UI	`data-state` with a word (`active`, `open`, `closed`)	`data-[state=active]:…` (arbitrary form: the value matters)
Headless UI 2.2	boolean attributes: `data-selected`, `data-open`, `data-hover`, `data-focus`	`data-selected:…` (Tailwind's boolean `data-*` shorthand, no brackets)
React Aria Components 1.22	`data-selected`, `data-hovered`, `data-focused` (and `data-pressed`, used in my build, not clicked)	`data-selected:…`, `data-hovered:…`

A CLASS THAT LOOKS RIGHT AND GENERATES NOTHING

My first React Aria attempt used `selected:bg-blue-500` and `hovered:underline`. Tailwind has no such variants, so the build produced no rule for them and no error. (React Aria publishes a separate Tailwind plugin that adds names like that; I did not install or test it.) Without it, use `data-selected:`. After wiring any library, search the built CSS for one of its classes.

What you get for free, from the Radix dialog test: clicking the trigger rendered the dialog into `<body>` (a portal) with `role="dialog"`, focus moved into the dialog (to its Close button), the page behind got `data-aria-hidden`, and after closing, the dialog was gone from the DOM and focus returned to the trigger. That is the part you do not want to write yourself. Keyboard operation and screen-reader output I did not test.

Choosing: all three worked with Tailwind in a Vite build (the page's CSS was 9.19 kB with all three libraries' examples). Radix is widely copied; Headless UI is made by the Tailwind team and also has a Vue edition (not run here); React Aria has a separate hooks API. I did not benchmark or audit any of them, so pick on the components you need and read each one's accessibility notes.

shadcn/ui: Install Once, Own the Code

CSS Frameworks 7 explains the idea: the components are not a dependency, the CLI copies their source into your project. I made a Vite + React + TypeScript project with Tailwind, the `@/` path alias, and ran `npx shadcn@latest init -d -y` (CLI 4.21.4). It created `components.json`, edited `src/index.css`, and wrote two files:

```
// src/lib/utils.ts
export { cn } from "cn"

// src/components/ui/button.tsx   (the top and bottom, middle shortened)
import { Button as ButtonPrimitive } from "@base-ui/react/button"
import { cva, type VariantProps } from "class-variance-authority"
import { cn } from "cn"

const buttonVariants = cva("group/button inline-flex shrink-0 items-center ...", {
  variants: {
    variant: { default: "bg-primary text-primary-foreground hover:bg-primary/80", outlin
    size: { default: "h-8 gap-1.5 px-2.5 ...", xs: "...", sm: "...", lg: "...", icon: "s
  },
```

```
    defaultVariants: { variant: "default", size: "default" },
  })

  function Button({ className, variant = "default", size = "default", ...props }) {
    return <ButtonPrimitive data-slot="button" className={cn(buttonVariants({ variant, siz
  }
```

```
components.json:  "style": "base-nova",  "cssVariables": true,  aliases "@/components",
dependencies added: @base-ui/react, class-variance-authority, cn, lucide-react, tw-anima
NOT added: clsx, tailwind-merge
```

Everything in this chapter is in those two files: `cva` for variants, `cn` to merge, a headless primitive (here Base UI, not Radix: that is what this CLI version's default style uses) for behaviour, and semantic colour tokens (`bg-primary`, `text-primary-foreground`) that chapter 2's `@theme inline` pattern defines in `index.css`. The surprise was `utils.ts`: I expected the familiar `twMerge(clsx(inputs))`, but it is one line re-exporting `cn` from an npm package named `cn`. I checked it: version 0.4.0, repository `shadcn-ui/cn`, maintainer shadcn, described as a drop-in replacement for clsx and tailwind-merge. Run 19 of my merge inputs through both libraries and they gave identical output; the package's README claims the same for 356,000 differential tests and large speed-ups, which I did not measure. Its custom-theme API is `createCn`, the equivalent of `extendTailwindMerge` (I ran the same `text-huge` example through it and it resolved the conflict).

Then I built a page with four buttons and measured what the component does with a caller's `className`:

```
<Button id="d">Default</Button>
<Button id="w" className="px-8">px-8</Button>
<Button id="g" className="bg-green-600 hover:bg-green-700">Green</Button>
<Button id="o" variant="outline">Outline</Button>
```

```
d   bg-primary hover:bg-primary/80 px-2.5        padding-left 10px   background oklch(0.
w   bg-primary hover:bg-primary/80 px-8          padding-left 32px   px-2.5 was removed
g   bg-green-600 hover:bg-green-700 px-2.5       bg-primary and hover:bg-primary/80 remo
o   bg-background hover:bg-muted px-2.5          background oklch(1 0 0)
build: 23.56 kB CSS, 257 kB JavaScript (83 kB gzip)
```

The caller's classes replaced the matching ones, including the `hover:` pair, because here the caller overrode both. The file is yours: you can edit it, and nothing updates it behind your back. That is the trade: no dependency to upgrade, but every upstream fix is a merge you perform yourself.

daisyUI and Flowbite: Classes Instead of Components

The other approach ships *class names* (`btn btn-primary`) and the CSS behind them, as a Tailwind plugin. I installed both and measured what they cost on small pages.

	DAISYUI 5.7.47	FLOWBITE 4.0.2
Setup in CSS	`@plugin "daisyui";`	`@import "flowbite/src/themes/default"; @plugin "flowbite/plugin"; @source "./node_modules/flowbite";`
Behaviour	none in what I ran: CSS only	a JavaScript file (`flowbite.min.js`) reads `data-*` attributes
Theming	themes by `data-theme`: `@plugin "daisyui" { themes: light --default, dark --prefersdark, cupcake; }` produced rules for `[data-theme=light]`, `dark` and `cupcake`	a theme file (`default`, `minimal`, …) defines tokens such as `--color-brand` (`var(--color-blue-700)` in the default theme), used as `bg-brand`
CSS with an empty page	10,998 B (plain Tailwind: 4,165 B)	not measured
CSS with one small page	22,744 B (default themes), 23,843 B with three themes; plain Tailwind 6,151 B for that page	26,725 B without the `@source` line, **57,882 B** with it; plain Tailwind 10,564 B
Overriding	utilities win: `btn btn-primary px-12` gave padding 48px (default 16px) and `bg-red-500` replaced the primary background	ordinary Tailwind classes in your markup

The sizes are minified output for tiny pages, so treat them as a rough scale, not a benchmark: a styling library costs kilobytes before you write a line. The override result matters more. daisyUI's rules sit in their own cascade layer (`daisyui`, with sub-layers), beneath Tailwind's `utilities`

layer (chapter 1), so a utility beats a component class. That is why `btn px-12` works without a merge helper.

Flowbite's `@source` line scans the entire library for class names, which is why that build grew to 57,882 B. The dropdown test showed the second difference: with the script loaded, clicking a button with `data-dropdown-toggle="dd"` changed the panel from `hidden` (display: none) to `block`. I did not test Flowbite without the script, its React or Vue packages, daisyUI's interactive components, or either library's accessibility.

When a Plain Component Is Enough

For a small product, a lookup table and a join can do the job of `cva`:

```
const sizes   = { sm: 'px-2 py-1 text-sm', md: 'px-4 py-2 text-base' };
const intents = { primary: 'bg-blue-600 text-white', danger: 'bg-red-600 text-white' };

const Button = ({ intent = 'primary', size = 'md' } = {}) =>
  ['inline-flex rounded-md', intents[intent], sizes[size]].join(' ');
```

```
Button()                 inline-flex rounded-md bg-blue-600 text-white px-4 py-2 text-b
Button({ size: 'lg' })   inline-flex rounded-md bg-blue-600 text-white        (unknown
with "as const" in TypeScript:  plain["lg"]  ->  error TS7053: ... Property 'lg' does no
```

The output matches the `cva` version, including the silent unknown value, and TypeScript catches the typo in both. What `cva` adds is compound variants, defaults in one place, and `VariantProps` deriving the prop types from the table. What the plain version lacks is exactly what a caller's `className` needs: conflict handling. A plain component can avoid the need by not accepting arbitrary classes at all (offer props instead). Once callers pass `className`, add `cn`.

SITUATION	REASONABLE CHOICE	WHY
One or two components, a few variants	plain lookup table, no extra packages	nothing to conflict; the table is the documentation
A design system, many variants, callers add classes	`cva` + `cn`	typed variants, one merge point

SITUATION	REASONABLE CHOICE	WHY
Interactive widgets (tabs, dialogs, menus)	a headless library	focus and ARIA are the hard part; I saw the dialog's focus handling work, keyboard and screen readers untested
Want finished-looking components you can edit	shadcn/ui	you own the files; upstream fixes are your merges
Want finished looks with the least markup	daisyUI	class names, themes by attribute; utilities still override
Want ready widgets plus a script, any back end	Flowbite	`data-*` attributes and one script; check the `@source` cost

MISTAKES WHILE PREPARING THIS CHAPTER

1. I used `selected:` and `hovered:` as if they were Tailwind variants. The build raised no error and wrote no rule. **How to avoid it:** after adding a library's classes, search the built CSS for one of them before testing in the browser; a silent absence is the usual failure.

2. I expected shadcn's `utils.ts` to contain `twMerge(clsx(…))`. That is what older tutorials (and my memory) show; the CLI I ran generated a re-export from a different package. **How to avoid it:** read the files the tool wrote instead of describing them from memory, and look up an unfamiliar dependency (owner, repository, version) before relying on it.

3. My plain-component test crashed when called with no arguments. A destructured parameter needs a default (`= {}`). **How to avoid it:** run the no-argument call first; it is the most common call.

4. I wrote a `tsconfig.json` with `baseUrl`, and TypeScript 7 rejected it (`TS5102: Option 'baseUrl' has been removed`). **How to avoid it:** when a tool's major version changes, read its first error message; this one named the replacement.

Hands-On Exercises

EXERCISE 1

Render four elements with the conflicting classes `bg-red-500 bg-blue-500` and `px-8 px-2` in both attribute orders and report which wins. Then run ten class pairs through `twMerge`, include a custom font size from `@theme`, show that it is not merged, and fix it with `extendTailwindMerge`.

View solution

EXERCISE 2

Write a `cva` button with two variant groups, a compound variant and defaults. Call it five ways (defaults, both set, an unknown value, an extra `class`, wrapped in `cn`), show the `hover:` leftover when a caller overrides the background, and show the TypeScript error for the unknown value.

View solution

EXERCISE 3

In a Vite + React build, style a tab list with one headless library using its state attributes. Click the second tab in headless Chrome and report the attributes and background of both tabs before and after. Then use a variant name that does not exist and show that the CSS has no rule for it.

View solution

CHAPTER 6 QUICK REFERENCE

- Conflicting utilities are decided by the order of rules in the built CSS, not by the order in the `class` attribute; `twMerge` removes the loser from the string beforehand
- `cn = (...a) => twMerge(clsx(a))`. `clsx` joins, `twMerge` resolves conflicts; different variants (`hover:bg-*` and `bg-*`) do not conflict
- Custom `@theme` names in size, shadow or radius groups are not known to `twMerge`: use `extendTailwindMerge` (the colour case happened to work)
- `cva(base, { variants, compoundVariants, defaultVariants })` builds a string only; unknown values add nothing silently; `VariantProps` makes them a type error; merge with `cn(button(...), className)`
- Headless state attributes: Radix `data-[state=active]:`; Headless UI `data-selected:` `data-open:` `data-hover:`; React Aria `data-selected:` `data-hovered:`. A made-up variant builds without error and writes no CSS
- shadcn/ui (CLI 4.21.4, style `base-nova`): copies `button.tsx` and `utils.ts` into your project; `cn` comes from the `cn` package; you own the files
- daisyUI 5: `@plugin "daisyui";`, themes by `data-theme`, utilities override its classes. Flowbite 4: theme import + `@plugin "flowbite/plugin"` + `@source` + a script; the `@source` line made my small

page's CSS 2.2 times larger

- Plain components: a lookup table and a join; give callers props, not `className`, or add `cn`

COMING NEXT

Chapter 7 is about keeping a Tailwind project healthy: what affects output size and build time, the Prettier plugin that sorts classes, ESLint and editor tooling, a code-review checklist, visual-regression testing and browser-support decisions. Start with *Tailwind CSS Intermediate/Advanced 7*.

Tailwind CSS Intermediate/Advanced

Chapter 7 · Performance, Tooling and Maintenance

A Tailwind project is easy to start and, after a year, easy to let go untidy: class lists in different orders, typos that silently do nothing, conflicting classes, a stylesheet nobody has looked at. This chapter measures the things people worry about (output size, build time), then sets up the tools that keep a project tidy (Prettier, ESLint, editor settings), and finishes with a way to catch visual changes and a review checklist.

> **EVERY EXAMPLE HERE WAS RUN, EXCEPT WHERE MARKED**
>
> Tailwind 4.3.3, Node 24.18.0, Windows 11, Chrome 154 headless. Tools: prettier 3.9.10 with prettier-plugin-tailwindcss 0.8.1, eslint 10.12.0 with eslint-plugin-tailwindcss 4.4.0 and eslint-plugin-better-tailwindcss 4.9.0, @typescript-eslint/parser 8.71.1, pixelmatch and pngjs. Timings come from one Windows laptop and a synthetic project, so they show proportions, not what your machine will do.
> **Not run:** the VS Code extension (its settings are from its README), the Tailwind release notes' speed figures, Safari and Firefox, other bundlers' build times, and a real CI pipeline.

What Decides the Size of the Output

The generated CSS contains one rule per distinct class that the scanner found (chapter 1), plus the reset and the theme variables you use. It does *not* grow with the number of files or with how often a class repeats. I generated three projects of 500 files each, with 30 random classes per file, drawn from pools of different sizes, and built each from inside its own folder:

PROJECT	DISTINCT CLASSES	PLAIN	`--minify`	MINIFIED + GZIP
small pool	150	18,992 B	13,605 B	3,072 B
colours, spacing, sizes	785	53,825 B	41,673 B	7,050 B
the same 785 with 5 variant prefixes (`hover:`, `md:`, `dark:`, `focus:`, `lg:hover:`)	4,513	326,471 B	239,489 B	29,008 B

Size follows the number of *distinct* class-plus-variant combinations. Every extra variant on a class is another rule. A page that is 30 KB gzipped after minifying has thousands of distinct

combinations, not thousands of elements. Separately, three settings changed the size of the 150-class project (13,605 B minified, 3,062 B gzipped):

```
@import "tailwindcss";                                          /* default */
@import "tailwindcss" theme(static);                            /* write every theme va
@layer theme, base, components, utilities;
@import "tailwindcss/theme.css" layer(theme);                   /* no reset (preflight)
@import "tailwindcss/utilities.css" layer(utilities);
```

```
default                          13,605 B   gzip 3,062 B
theme(static)                    27,524 B   gzip 6,647 B     every variable written whet
theme + utilities only           9,953 B    gzip 1,907 B     no preflight: the reset is
default, --optimize, no minify   17,309 B                      (--optimize is not --mini
```

Two honest limits. First, I did not test whether dropping the reset is safe for your pages; the saving (3.6 KB here) is the reset's cost, and the browsers will then apply their own default margins and fonts. Second, the build already writes only the theme variables you use, so `theme(static)` is the expensive direction; use it only when something outside Tailwind reads the variables.

Build Time

On the 4,513-class project (500 files) I first timed `npx @tailwindcss/cli …` and got about 3.5 seconds for every project, whatever its size. That was `npx` starting up. Calling the program directly (`node node_modules/@tailwindcss/cli/dist/index.mjs`) and reading the CLI's own "Done in" line gave a very different picture:

```
cold build, 4,513 classes, 500 files:  Done in 225 ms, 328 ms, 218 ms     (three runs)
cold build, 150 classes, 500 files:    Done in 120 ms, 121 ms, 251 ms

watch mode (--watch=always), same large project, one edit at a time:
  start-up                                  356 ms
  append a class to a content file (x3)     lines of 5, 48, 4, 69, 6, 33, 82, 66 ms a
  add an arbitrary-value class (text-[#abc123])   included in the lines above; output co
  edit the CSS file (add a @theme colour)       270 ms; the new bg-zz class appeared in
```

Watch mode printed several "Done" lines for each edit (the file system reports more than one event), so I cannot say which line belongs to which edit; the range is what I can state. Content edits were from under a millisecond to about 80 ms; changing the CSS file took 270 ms, close to a cold build, because the theme may have changed. I did not verify the speed-up figures in Tailwind's release notes, and I did not time bundler integrations (Vite, Next.js). If a build is slow, measure the tool itself with its own timer before blaming Tailwind.

Browser Support Is a Decision

Tailwind's compatibility page says v4 "is designed for and tested on modern browsers" and depends on **Chrome 111** (March 2023), **Safari 16.4** (March 2023) and **Firefox 128** (July 2024). The page does not say which features set those numbers. I built a page with a shadow, a ring, a translate, a rotate, an opacity colour, a hover colour and a backdrop blur and counted what the CSS uses:

```
@layer              6     cascade layers (chapter 1)
@property           26    registered custom properties (--tw-translate-x, --tw-shadow,
@supports           4     feature tests around newer syntax
color-mix           5     opacity on theme colours, placeholder colour
@media (hover       1     hover: only where a hover pointer exists
oklch               3     colours (the full palette has about 50 in a small build)
```

That is the list your decision rests on: layers, `@property`, `color-mix()` and `oklch()`. If a real share of your visitors use an older browser, Tailwind v4 is the wrong choice and staying on v3 is the supported one (chapter 8 covers the migration and when not to). Check your own analytics, not a general rule. Tailwind's page also says the bleeding-edge utilities (`field-sizing`, `@starting-style`, `text-wrap: balance`) are optional: do not use them if your targets lack them. I did not test any browser except Chrome.

Sorting Classes with Prettier

A class list in a different order on every element makes diffs noisy and reviews slower. `prettier-plugin-tailwindcss` sorts them into Tailwind's own order. In v4 it needs to know your CSS file, so it can see custom theme names. My configuration and results:

```
// .prettierrc
{
  "plugins": ["prettier-plugin-tailwindcss"],
  "tailwindStylesheet": "./app.css",
  "tailwindFunctions": ["cn", "cva"]
}

/* app.css */
@import "tailwindcss";
@theme { --color-brand: oklch(55% .2 260); }
```

```
before: px-4 hover:bg-brand text-white md:px-8 bg-brand rounded-md flex font-bold p-2 cu
after:  custom-thing flex rounded-md bg-brand p-2 px-4 font-bold text-white hover:bg-bra

without "tailwindStylesheet" (same input):
        hover:bg-brand bg-brand custom-thing flex rounded-md p-2 px-4 font-bold text-whi
```

The order is: classes Tailwind does not know first (`custom-thing`), then utilities in Tailwind's order, then variants (`hover:`, `md:`, `dark:`) last. Without the stylesheet setting the plugin did not know `bg-brand` (a theme name) and put it in the unknown group at the front, so a missing setting changes the sort silently. For helper functions:

```
export const B = () => <div className={cn("p-4 flex mt-2 items-center", x && "text-white
export const v = cva("p-4 flex mt-2", { variants: { s: { a: "text-lg px-2 flex" } } });
```

```
with "tailwindFunctions": ["cn", "cva"]:
  className={cn("mt-2 flex items-center p-4", x && "bg-red-500 p-1 text-white")}
  cva("mt-2 flex p-4", { variants: { s: { a: "flex px-2 text-lg" } } })

without it:
  className={cn("mt-2 flex items-center p-4", x && "bg-red-500 p-1 text-white")}     (st
  cva("p-4 flex mt-2", { variants: { s: { a: "text-lg px-2 flex" } } })              (NO
```

Strings inside a `className` attribute are sorted without any setting, even inside `cn(...)`. Strings in a call that is *not* inside a class attribute, like the `cva` table, need `tailwindFunctions`.

Linting: Typos, Conflicts and Unknown Classes

A typo like `bg-brandd` builds without error and styles nothing (chapter 6). A linter can catch it. I ran two ESLint plugins on the same file with a conflict, a typo and an unknown class:

```
<div className="p-4 p-2 flex block text-red-500 bg-brandd unknown-thing mt-2">

// eslint-plugin-tailwindcss 4.4.0   settings: { tailwindcss: { cssConfigPath: './app.cs
  error    'p-4' conflicts with 'p-2'                              tailwindcss/no-contrad
  error    'flex' conflicts with 'block'                           tailwindcss/no-contrad
  warning  Classname 'bg-brandd' is not a Tailwind CSS class       tailwindcss/no-custom-
  warning  Classname 'unknown-thing' is not a Tailwind CSS class   tailwindcss/no-custom-
  warning  Invalid Tailwind CSS classnames order                   tailwindcss/classnames

// eslint-plugin-better-tailwindcss 4.9.0   settings: { 'better-tailwindcss': { entryPoi
  error    Conflicting class detected: "p-4" and "p-2" apply the same CSS properties: "p
  error    Conflicting class detected: "flex" and "block" apply the same CSS properties:
  error    Unknown class detected: bg-brandd
  error    Unknown class detected: unknown-thing
  warning  Incorrect class order ...      warning  Incorrect line wrapping ...
```

Both work with v4 and both read your CSS file so that `bg-brand` counts as known (the typo `bg-brandd` does not). They differ in what they report as an error: the first treats an unknown class as a warning (it may be your own class), the second as an error. Plugins change faster than Tailwind: `eslint-plugin-tailwindcss` 4.4.0 exposes its config as `configs.recommended` only; the `"flat/recommended"` name from its README example (written as `flat/recommended || recommended`) did not exist in this version.

Do the tools agree with each other? I formatted a line with Prettier and linted the result. `eslint-plugin-tailwindcss` reported nothing. `eslint-plugin-better-tailwindcss` reported its order rule satisfied, but its *line-wrapping* rule wanted the classes split over several lines, which fights Prettier. If you use Prettier, turn that rule off (`"better-tailwindcss/enforce-consistent-line-wrapping": "off"`; I did not run that exact setting) and let one tool own each job.

Editor settings (VS Code)

Not run here; these come from the Tailwind CSS IntelliSense extension's README. `"files.associations": {"*.css": "tailwindcss"}` opens CSS files in Tailwind mode. `tailwindCSS.classAttributes` (default `class`, `className`, `ngClass`, `class:list`) and

`tailwindCSS.classFunctions` say where to offer completions; add `cn` and `cva` there as well as in Prettier. `tailwindCSS.lint.cssConflict` (default warning) flags classes that set the same property, and `tailwindCSS.lint.suggestCanonicalClasses` flags class names not in their shortest form.

Long Class Lists

A button with fifteen classes repeated on every button is a maintenance problem, not a size problem. I built 200 buttons with the same 15 classes inline, and again with a `.btn` class made by `@apply` inside `@layer components`:

```
inline (15 classes on each of 200 buttons)  HTML 48,290 B (gzip 824)   CSS 7,204 B (gzip
.btn via @apply                              HTML  7,690 B (gzip 547)   CSS 6,984 B (gzi
```

Repetition compresses almost to nothing, so the transferred difference was 355 bytes (12%) on 200 identical buttons. The real argument for extracting is that you change the button in one place. The options, in the order I would try them:

OPTION	USE WHEN	COST
A component (React, Vue, Astro, a template partial)	the same markup repeats	none beyond the component; the classes stay in one file (chapter 6)
Variants with `cva`	the repeated thing has options	a dependency; unknown values fail silently without types
`@apply` in `@layer components`	plain HTML or a CMS with no component system	a second place to look; utilities still override it because of layer order
Leave it long	the list is used once	nothing; a long list used once is fine

Catching Visual Changes

A class change can break a page without any test failing. The cheapest check is a screenshot comparison. I rendered a small card page in headless Chrome (`--screenshot`), took screenshots of the same page, and compared them with `pixelmatch`:

```
v1a vs v1b  the same page screenshotted twice (new browser profile each time)     0 diff
v1a vs v2   button bg-blue-600 -> bg-blue-500 and py-2 -> py-2.5                  3,253
v1a vs v3   text-slate-600 -> text-slate-500 on one paragraph
             pixelmatch threshold 0.1 (default)                                   185
             pixelmatch threshold 0                                               346
```

On one machine with the same browser, two runs of an unchanged page were identical, so any difference was a real change. A one-shade colour change was caught at both thresholds. Two cautions that I did *not* test: screenshots from a different operating system or browser version will differ in text rendering, so compare only against screenshots from the same environment (usually your CI image), and a threshold that is too loose will hide small changes like the shade above. Tools such as Playwright's screenshot comparison automate this; I did not run it.

A Review Checklist for Tailwind Markup

- **Complete class names.** No `'bg-' + color` (chapter 1). Variant maps contain whole classes.
- **Conflicts.** No two classes setting the same property on one element; component `className` goes through `cn` (chapter 6), and a `hover:` counterpart is overridden too.
- **Tokens, not literals.** A colour or size used in more than one place is a theme token (chapter 2), not `bg-[#3b82f6]` repeated.
- **State uses the right attribute.** `data-*` or `aria-*` variants match the library's real attributes (chapter 4 and 6); grep the built CSS if unsure.
- **Sorted and linted.** Prettier and ESLint pass with no per-file exceptions.
- **Length.** A class list repeated three times becomes a component or variant.
- **Browser targets.** No bleeding-edge utility the targets lack.
- **Visual check.** The screenshot diff for changed components was looked at, not just passed.

MISTAKES WHILE PREPARING THIS CHAPTER

1. My first size comparison gave identical results for three different projects. I ran the CLI from the parent folder, so the scanner (which starts at the working directory) read all three. **How to avoid it:** when results that should differ come out the same, check what was scanned; this is the chapter 1 lesson again.

2. I timed `npx`, not the build. Every run took about 3.5 seconds; the build itself took 120 to 330 ms.

How to avoid it: time the program directly and compare with its own timer.

3. `--watch` **exited immediately without watching.** In a non-interactive shell it needs `--watch=always`. **How to avoid it:** look for the first "Done" line in the log before editing any file.

4. A test that could not show a difference. I limited the scan to a glob that, as it happened, matched 111 of 500 files, which together contained every one of the 150 classes, so the size was unchanged. I dropped it. **How to avoid it:** work out in advance what result would prove the claim, and build the input so that result is possible.

5. I copied the plugin's README config and it failed (`configs["flat/recommended"] is not iterable`). **How to avoid it:** print `Object.keys(plugin.configs)` for the version you installed.

Hands-On Exercises

EXERCISE 1

Generate three 500-file projects with different numbers of distinct classes (and one with variant prefixes), build each from inside its own folder, and report plain, minified and gzipped sizes. Then time the same build with `npx` and by calling the CLI directly, and measure one `--watch=always` rebuild.

View solution

EXERCISE 2

Configure Prettier with the Tailwind plugin and ESLint with one of the two Tailwind plugins. Run them on a file with a conflict, a typo and an unsorted list, show the report, fix it, and show that Prettier and ESLint then agree. Include a custom theme colour and a `cva` call.

View solution

EXERCISE 3

Screenshot a small page with headless Chrome, change one class, screenshot again and count differing pixels with `pixelmatch`. Include the unchanged-twice control and compare two thresholds on a subtle colour change.

View solution

CHAPTER 7 QUICK REFERENCE

- Output size follows *distinct* class-plus-variant combinations, not files or repeats: 150 classes 13.6 KB, 785 classes 41.7 KB, 4,513 classes (with variants) 239 KB minified; gzipped 3.1, 7.1 and 29.0 KB
- `theme(static)` doubled the 150-class output; dropping preflight saved 3.6 KB but removes the reset; `--minify` shrinks, `--optimize` does not
- Time the program, not `npx`. Cold build of 500 files: 120 to 330 ms; watch edits mostly under 100 ms; CSS edits about a cold build. Use `--watch=always` in scripts
- v4 needs Chrome 111, Safari 16.4, Firefox 128 (Tailwind's compatibility page); the output uses layers, `@property`, `color-mix()`, `oklch()`
- Prettier: `plugins`, `tailwindStylesheet` (needed to know theme names), `tailwindFunctions` (needed for `cva`); strings inside `className` are sorted anyway
- ESLint: `eslint-plugin-tailwindcss` (`cssConfigPath`) or `eslint-plugin-better-tailwindcss` (`entryPoint`); both caught conflicts and the typo. Turn off the second one's line-wrapping rule if you use Prettier
- Extract repeated markup into components or variants for maintenance; the transfer saving was 12% gzipped
- Screenshots from the same browser and machine were byte-identical, so a pixel diff is meaningful; keep references per environment

COMING NEXT

Chapter 8 migrates a project from Tailwind v3 to v4: the upgrade tool, the new import line, moving the JavaScript config into CSS, renamed and removed utilities, changed defaults, running both versions side by side, and a worked migration of a small project. Start with *Tailwind CSS Intermediate/Advanced 8*.

Tailwind CSS Intermediate/Advanced

Chapter 8 · Migrating from v3 to v4

Many real Tailwind projects are still on v3. The official upgrade tool does most of the mechanical work, but "most" is the problem: what it leaves behind builds without errors and looks slightly wrong. This chapter migrates a small v3 project with the real tool, measures exactly what is different afterwards (computed styles and a screenshot comparison, from chapter 7), finishes the job by hand until the page is pixel-identical, and lists the renamed, removed and changed utilities with results from the v4 build, not just from the documentation.

> **EVERY EXAMPLE HERE WAS RUN, EXCEPT WHERE MARKED**
>
> Starting point: Tailwind 3.4.19 (the `v3-lts` tag on npm) with `@tailwindcss/typography` 0.5.20, PostCSS 8.5.29 and autoprefixer 10.6.1. Upgraded with `@tailwindcss/upgrade` to Tailwind 4.3.3, on Node 24.18.0 and Windows 11, checked in headless Chrome 154. Product claims about the upgrade (browser support, the list of removed utilities, changed defaults) are from Tailwind's upgrade guide, fetched today, and I tested each one I could. **Not run:** a large real project, the Vite and Next.js plugin switch (chapter 5 covers those packages), projects using `prefix`, `important` or `corePlugins` in the config, other plugins, and any browser except Chrome.

Decide First: Should You Migrate Now?

The upgrade guide says v4 is "designed for Safari 16.4+, Chrome 111+, and Firefox 128+" and that "if you need to support older browsers, stick with v3.4 until your browser support requirements change." That is the one real blocker (chapter 7). The other question is time. The endoflife.date listing (a third-party tracker, not Tailwind's own announcement, so confirm it) gives the 3.4 line an end-of-life of **2027-02-28**; today is 2026-10-12, so that is about four and a half months away. The same listing shows 3.4.19 (released 2025-12-10) as the last 3.4 release. Staying on v3 is a choice with a visible end date.

The Project to Migrate

I wrote a deliberately awkward v3 project: a JavaScript config with extended colours, font, screen and radius, a safelist, class-based dark mode and the typography plugin; CSS with `@tailwind`

directives, `@apply`, `@layer components` and the `theme()` function; and a page that uses classes whose meaning changed in v4.

```
/** @type {import('tailwindcss').Config} */
module.exports = {
  darkMode: 'class',
  content: ['./index.html', './src/**/*.{html,js}'],
  safelist: ['bg-emerald-500','bg-pink-' + '500'],
  theme: {
    extend: {
      colors: { brand: { 500: '#4f46e5', 600: '#4338ca' }, surface: '#f1f5f9' },
      fontFamily: { display: ['Georgia', 'serif'] },
      screens: { '3xl': '1920px' },
      borderRadius: { card: '0.75rem' },
    },
  },
  plugins: [require('@tailwindcss/typography')],
};

/* src/input.css */
@tailwind base;
@tailwind components;
@tailwind utilities;

@layer base {
  h1 { @apply text-2xl font-bold; }
}

@layer components {
  .btn { @apply inline-flex items-center rounded-md bg-brand-500 px-4 py-2 text-sm font-
  .panel { background: theme('colors.surface'); border-radius: theme('borderRadius.card'
}
```

```
<!doctype html>
<html lang="en"><head><meta charset="utf-8"><link rel="stylesheet" href="dist/out.css"></
<body class="bg-white p-6 font-sans text-slate-800 dark:bg-slate-900">
  <div class="panel shadow max-w-md">
    <h1 class="font-display">Migration demo</h1>
    <p class="mt-2 text-sm text-slate-600">A small page using several things that change
    <div class="mt-4 flex space-x-2">
      <button class="btn">Save</button>
      <button class="btn bg-opacity-50 bg-black" id="b2">Faded</button>
```

```
      <button class="rounded shadow-sm ring border px-3 py-2 outline-none flex-grow" id=
    </div>
    <div class="mt-4 h-8 bg-gradient-to-r from-brand-500 to-emerald-500 blur-sm"></div>
    <p class="mt-4 overflow-ellipsis !font-bold text-brand-600 3xl:text-xl">Important an
    <div class="mt-4 transform scale-105 rounded-card bg-emerald-500 p-2 text-white">saf
    <article class="prose mt-4"><h2>Prose</h2><p>Typography plugin text.</p></article>
  </div>
</body></html>
```

```
baseline: the page built with Tailwind 3.4.19, screenshot v3.png (700 x 700), and a scri
for 14 things (border colour, ring, outline, background opacity, panel background, space
```

Run the Upgrade Tool

The upgrade tool is run with `npx @tailwindcss/upgrade`. Tailwind's guide says it needs Node 20 or higher and recommends a new branch. It also checks git: on a copy of the project with one uncommitted edit it refused:

```
npx @tailwindcss/upgrade
```

```
│ Git directory is not clean. Please stash or commit your changes before migrating.
│ You may use the `--force` flag to silence this warning and perform the migration.
```

On the committed project it ran for about 25 seconds and reported:

```
Upgrading from Tailwind CSS v3.4.19
Linked .\tailwind.config.js to .\src\input.css
Migrating JavaScript configuration files…
  The configuration file at .\tailwind.config.js could not be automatically migrated to
  format, so your CSS has been updated to load your existing configuration file.
Migrated stylesheet: .\src\input.css
Updated package: tailwindcss
Migrated .\index.html     Migrated templates for configuration file: .\tailwind.config.j
Installed package: @tailwindcss/postcss
Migrated PostCSS configuration: .\postcss.config.js
Verify the changes and commit them to your repository.
```

The tool changed five files (`index.html`, `package.json`, `package-lock.json`, `postcss.config.js`, `src/input.css`) and did *not* convert the JavaScript config; it told me so, without saying why, and loaded it with `@config` instead. The changes that matter:

```
- @tailwind base;  @tailwind components;  @tailwind utilities;
+ @import 'tailwindcss';
+ @config '../tailwind.config.js';
+ @layer base { *, ::after, ::before, ::backdrop, ::file-selector-button { border-color:

- .btn { @apply ... focus:outline-none focus:ring-2; }         (inside @layer components
+ @utility btn { @apply ... focus:outline-hidden focus:ring-2; }
- .panel { background: theme('colors.surface'); border-radius: theme('borderRadius.card'
+ @utility panel { background: var(--color-surface); border-radius: var(--radius-card);

index.html:   shadow -> shadow-sm   |   shadow-sm -> shadow-xs   |   rounded -> rounded-
              flex-grow -> grow   |   bg-gradient-to-r -> bg-linear-to-r   |   blur-sm -
postcss.config.js:  tailwindcss: {}  ->  '@tailwindcss/postcss': {}      package.json: t
```

Look at the order of the renames: `shadow` became `shadow-sm` *and* `shadow-sm` became `shadow-xs`. The names shifted down one step, so a careless find-and-replace would turn every old `shadow-sm` into the new, larger `shadow-sm`. The tool handles that ordering; a hand edit must too.

What the Tool Left Wrong

I built the upgraded project (it built: exit code 0, no warnings) and compared it with the v3 baseline in Chrome, once by computed styles and once by screenshot (`pixelmatch`, threshold 0.1):

```
v3.png vs after-the-tool.png:   4,279 differing pixels of 490,000

computed styles that differ (v3 -> after the tool):
  #b3 ring:      box-shadow ring  rgba(59,130,246,0.5) 3px   ->   3px ring in the TEXT c
  #b2 faded:     background rgba(0,0,0,0.5)                  ->   rgb(0,0,0)
  .panel:        background rgb(241,245,249), radius 12px    ->   transparent, radius 0
  space-x-2:     second button margin-left 8px               ->   first button margin-r
```

```
border colour, gradient, colours:  same colour, now written in oklch/oklab (expected)
scale-105 with "transform": transform matrix(1.05,...)      ->   transform none (the s
```

Three of those are real regressions the tool did not (or could not) fix, and none produced a message:

1. `bg-opacity-50 bg-black` was left in the markup. In v4 there is no rule for `bg-opacity-*` (I built it: no output), so the faded button became solid black. The replacement is `bg-black/50`.
2. **The ring.** v3's `ring` was 3px of 50% blue; `ring-3` keeps the width but the default colour is `ring-blue-500/50`.
3. **The panel.** The tool rewrote `theme('colors.surface')` to `var(--color-surface)`, but the config was still JavaScript and, as far as I could tell, no such CSS variable existed, so the background vanished and the radius became 0. This follows from the config not being migrated, which the tool had reported.

Finishing the Job by Hand

The remaining work is the part the tool declined: moving the JavaScript config into CSS. Each setting has a v4 home:

IN `TAILWIND.CONFIG.JS` (V3)	IN CSS (V4)
`theme.extend.colors.brand[500]`	`@theme { --color-brand-500: #4f46e5; }`
`fontFamily.display`	`--font-display: Georgia, serif;`
`screens['3xl']`	`--breakpoint-3xl: 1920px;`
`borderRadius.card`	`--radius-card: 0.75rem;`
`darkMode: 'class'`	`@custom-variant dark (&:is(.dark *));`
`plugins: [require('@tailwindcss/typography')]`	`@plugin '@tailwindcss/typography';`
`safelist: [...]`	`@source inline("bg-{emerald,pink}-500");` (the config's `safelist` is ignored, see below)

IN `TAILWIND.CONFIG.JS` (V3)	IN CSS (V4)
`content: [...]`	automatic detection, plus `@source` if needed (chapter 1)
`theme('colors.x')` in CSS	`var(--color-x)`; spacing: `--spacing(4)`

I deleted `tailwind.config.js`, wrote those into `input.css`, and fixed the three markup problems:

```
@import 'tailwindcss';
@plugin '@tailwindcss/typography';
@source inline("bg-{emerald,pink}-500");
@custom-variant dark (&:is(.dark *));

@theme {
  --color-brand-500: #4f46e5;
  --color-brand-600: #4338ca;
  --color-surface: #f1f5f9;
  --font-display: Georgia, serif;
  --breakpoint-3xl: 1920px;
  --radius-card: 0.75rem;
}

@layer base {
  *, ::after, ::before, ::backdrop, ::file-selector-button { border-color: var(--color-g
  h1 { @apply text-2xl font-bold; }
}

@utility btn {
  @apply inline-flex items-center rounded-md bg-brand-500 px-4 py-2 text-sm font-medium
}
@utility panel {
  background: var(--color-surface);
  border-radius: var(--radius-card);
  padding: --spacing(4);
}

<!-- index.html changes -->
<button class="btn bg-black/50" id="b2">Faded</button>
<button class="rounded-sm shadow-xs ring-3 ring-blue-500/50 border px-3 py-2 outline-hid
<div class="mt-4 scale-105 rounded-card bg-emerald-500 p-2 text-white">safelisted</div>
```

```
v3.png vs after-the-manual-work.png:   0 differing pixels of 490,000
dark mode: <html class="dark"> still turns the body dark (rgb(15,23,42) in v3, the same
the pink safelisted class is generated (1 rule) by @source inline
```

The safelist trap

While testing, `bg-pink-500` (safelisted but never used in the markup) *did* appear in the v4 output with `@config` still loading the file, which looked like proof that `safelist` still works. It was not. The scanner reads every file it can see, including `tailwind.config.js`, and found the literal text `'bg-pink-500'`. When I wrote the entry as `'bg-pink-' + '500'` the v3 build still generated the class and the v4 build did not. Tailwind's guide agrees: "`corePlugins`, `safelist`, and `separator` ... are not supported in v4.0." Use `@source inline()` (chapter 1).

Renamed, Removed and Changed: Checked Against 4.3.3

I built a page containing both old and new names and printed the generated rule for each, so the table shows behaviour, not just the guide's list:

CLASS	V4.3.3 RESULT	MEANING
`bg-opacity-50`, `text-opacity-50`	**no rule**	removed; use `bg-black/50`. Silent: the element just loses the effect
`shadow` / `shadow-sm` / `shadow-xs`	`shadow` and `shadow-sm` are the same rule (1px 3px, the old `shadow`); `shadow-xs` is the old `shadow-sm`	old `shadow-sm` silently gets bigger if you do not rename it
`rounded` / `rounded-sm` / `rounded-xs`	`rounded` = 0.25rem; `rounded-sm` = `var(--radius-sm)`; `rounded-xs` exists	the same one-step shift as shadows
`blur` / `blur-sm` / `blur-xs`	all three exist	shifted like shadows
`ring`, `ring-3`	`ring` is 1px `currentcolor`; `ring-3` is 3px `currentcolor`	width 3px to 1px, colour blue-500 to currentcolor

CLASS	V4.3.3 RESULT	MEANING
`outline-none`, `outline-hidden`	`outline-none`: `outline-style: none`; `outline-hidden` adds a forced-colors fallback	use `outline-hidden` to keep the v3 meaning
`flex-grow`, `flex-shrink-0`, `overflow-ellipsis`, `decoration-slice`	**still produced a rule** in 4.3.3	the guide lists them as removed in 4.0; in 4.3.3 they still built. I do not know if that is intended, so rename them (`grow`, `shrink-0`, `text-ellipsis`, `box-decoration-slice`) as the tool did
`bg-gradient-to-r` / `bg-linear-to-r`	both exist	the tool renames to `bg-linear-to-r`
`!flex` / `flex!`	both produce `display: flex !important`	new position is the end; the old one still works
`bg-[--brand]` / `bg-(--brand)`	`background-color: --brand;` (invalid) / `var(--brand)`	old variable shorthand builds but is broken CSS; use `(--brand)`
`grid-cols-[1fr,2fr]` / `grid-cols-[1fr_2fr]`	`grid-template-columns: 1fr,2fr` (invalid) / `1fr 2fr`	commas are no longer turned into spaces

The guide also lists defaults that changed without any class being renamed. Those I observed: border colour (`currentcolor` instead of gray-200, restored by the tool's compatibility block), ring (above), and the outline. Those I took from the guide and did not test: placeholder colour (now the text colour at 50%), buttons using `cursor: default` (restore pointer with a base rule), `hover:` applying only where the device supports hover, `dialog` margins, the `hidden` attribute now winning over `flex`, variant stacking applying left to right (`first:*:pt-0` becomes `*:first:pt-0`), the removal of the `container` `center`/`padding` options, and prefix syntax (`tw:flex`). Search your project for each of them before calling the migration done.

Traps That Build Without Complaint

```
@tailwind base;
@tailwind components;
@tailwind utilities;              /* a v3 stylesheet fed to the v4 CLI, markup: class="
```

```
exit code 0, "Done in 344ms", output file 66 bytes, 0 rules for text-red-500

@import "tailwindcss";
.x { @apply bg-opacity-50 p-2; }
exit code 1   Error: Cannot apply unknown utility class `bg-opacity-50`

.y { color: theme(colors.red.500); width: theme(spacing.4); border-color: theme(borderCo
-> color: oklch(63.7% 0.237 25.331); width: 1rem; border-color: currentcolor        (the
```

The first is the one to remember: leaving the v3 directives in place does not fail, it produces an empty stylesheet. After any migration, build and check that one known class has a rule (chapter 6's rule again). An `@apply` of a removed utility, by contrast, fails loudly. The `theme()` function is still understood, although the guide recommends variables.

Running Both Versions Side by Side

During a gradual migration you may need v3 for one folder and v4 for another. npm can install the same package under two names. I installed both in one folder and built the same markup with each:

```
npm install tailwind3@npm:tailwindcss@3.4.19 tailwindcss@4 @tailwindcss/cli@4

node node_modules/@tailwindcss/cli/dist/index.mjs -i new/in.css -o new/out.css
node node_modules/tailwind3/lib/cli.js -c old/tw3.config.js -i old/in.css -o old/out.css
```

```
markup: <p class="shadow-sm rounded bg-red-500 p-4">

old/out.css (v3)  .shadow-sm { --tw-shadow: 0 1px 2px 0 rgb(0 0 0 / 0.05); ... }
new/out.css (v4)  .shadow-sm { --tw-shadow: 0 1px 3px 0 var(--tw-shadow-color, rgb(0 0 0
```

Both ran from one `package.json` and the same class name produced different shadows, which is exactly why the two should not process the same files. Give each its own input CSS and its own set of templates. I did not run a framework build with two versions at once.

A Migration Checklist

1. Decide: browser targets (Chrome 111, Safari 16.4, Firefox 128) and the 3.4 end-of-life date.
2. Commit everything; make a branch. Take a baseline: screenshots and a few computed styles of pages that matter.
3. Run `npx @tailwindcss/upgrade` (Node 20+). Read its output, including what it says it could not do.
4. Build and check the output has rules for known classes (an empty stylesheet exits 0).
5. Move the JavaScript config into CSS: `@theme`, `@plugin`, `@custom-variant`, `@source inline()`. Delete the config and the `@config` line.
6. Search for removed and shifted classes: `*-opacity-*`, `ring`, `outline-none`, `shadow-*`, `rounded-*`, `blur-*`, `[--var]`, commas in `grid-cols-[...]`.
7. Compare screenshots against the baseline; look at every difference.
8. Remove the compatibility border-colour block only after adding explicit border colours where needed.

MISTAKES WHILE PREPARING THIS CHAPTER

1. My own v3 baseline was wrong. I wrote `surface: 'theme(colors.slate.50)'` in the config; v3 did not resolve it and the panel was transparent. I only saw it in the first probe output. **How to avoid it:** read the baseline results before starting the migration; a migration cannot be compared with a baseline that is itself broken.

2. I thought `safelist` still worked under `@config`. The class appeared only because the scanner read the config file's text. **How to avoid it:** test a feature with input the scanner cannot see (a concatenated string), not the same literal it can.

3. My comparison script broke on renamed classes and compared the wrong property. It selected `p.overflow-ellipsis`, which no longer existed, and it read `margin-left` for `space-x`, which v4 moved to `margin-right` on the other element. **How to avoid it:** select by something that survives the change and compare what the user sees (positions, pixels) rather than one property.

4. My script that listed each class's rule reported false "NO RULE" lines for names with special characters (`!flex`, `bg-(--brand)`). **How to avoid it:** check every "missing" with a plain search before believing it.

5. I first committed `node_modules` into the scratch project's git history. I started again with a `.gitignore`. **How to avoid it:** write the ignore file before the first `git add`.

Hands-On Exercises

EXERCISE 1

Build a small v3 project (JavaScript config, `@tailwind` directives, `@apply`, `theme()`, and classes whose meaning changed). Take a screenshot and 10+ computed styles. Run the upgrade tool, rebuild, and list every difference with its cause. Include the dirty-repo refusal.

View solution

EXERCISE 2

Finish the migration by hand: move the config into CSS, fix the markup, and reach zero differing pixels. Then prove whether `safelist` still works under `@config` without letting the scanner read the class name.

View solution

EXERCISE 3

Build a page of old and new class names with Tailwind 4 and print the rule each one generates. Then test three traps: v3 directives in the v4 CLI, `@apply` of a removed utility, and the old `[--var]` syntax. Finally run v3 and v4 side by side with an npm alias.

View solution

CHAPTER 8 QUICK REFERENCE

- Decide on browsers first: v4 needs Chrome 111, Safari 16.4, Firefox 128. endoflife.date lists 3.4 as end-of-life 2027-02-28 (third-party; confirm)
- `npx @tailwindcss/upgrade`: Node 20+, refuses a dirty git tree (`--force` overrides), changes templates, CSS, PostCSS config and `package.json`; run it on a branch

- In my run it did **not** convert the JS config (it said so and used `@config`); that left a transparent panel, a solid-black "faded" button and a different ring: 4,279 pixels. The manual finish gave 0
- JS config to CSS: `@theme` variables, `@plugin`, `@custom-variant dark`, `@source inline()`; `safelist` under `@config` is ignored (the scanner can hide this)
- Names shifted one step: old `shadow-sm` is `shadow-xs`, old `shadow` is `shadow-sm`; same for `rounded` and `blur`. `ring` is 1px currentcolor; `ring-3` is the old width; `outline-none` to `outline-hidden`
- Removed with no rule and no warning: `bg-opacity-*` and its relatives. Use `bg-black/50`
- Silent breakage in 4.3.3: v3 `@tailwind` directives give a 66-byte stylesheet; `bg-[--brand]` gives invalid CSS (`bg-(--brand)` is right); `grid-cols-[1fr,2fr]` keeps the comma (use underscores)
- Side by side: `npm install tailwind3@npm:tailwindcss@3.4.19` beside v4 and keep their inputs separate
- Verify with a baseline: screenshots plus computed styles before and after

COMING NEXT

Chapter 9 covers accessibility, internationalisation and print: focus styles, `sr-only`, contrast and motion preferences, forced colours, logical properties and RTL, and `print:` styles. Start with *Tailwind CSS Intermediate/Advanced 9*.

Tailwind CSS Intermediate/Advanced

Chapter 9 · Accessibility, Internationalisation and Print

Tailwind gives you variants for the user's preferences and situation: `focus-visible:`, `motion-reduce:`, `contrast-more:`, `forced-colors:`, `rtl:`, `print:`. The variants are easy to write and easy to get wrong without noticing, because a page that ignores these needs still looks fine to its author. So this chapter does what the earlier ones did: it turns each preference on in a real browser and measures what happens. The course *Web Accessibility (a11y)* covers the standards and the reasoning in depth; this chapter is about the Tailwind side.

> **EVERY EXAMPLE HERE WAS RUN, EXCEPT WHERE MARKED**
>
> Tailwind 4.3.3, Node 24.18.0, Windows 11, headless Chrome 154 driven over the DevTools Protocol (real key presses and mouse clicks; `Emulation.setEmulatedMedia` for the preferences; `Accessibility.getFullAXTree` for what assistive technology is offered; `Page.printToPDF` for print).
> **Not run:** a real screen reader, a real Windows contrast theme (Chrome's emulated forced colours only), a real printer, an automated audit tool such as axe, and any browser except Chrome. Passing these checks does not make a page accessible; they show what each class does.

Focus Styles

Keyboard users need to see where they are. Three habits matter, and I tested each on four controls: a button using `focus:`, a button using `focus-visible:`, a button with `outline-hidden` and no replacement, and a text input using `focus-visible:`. I clicked each with the mouse, then pressed Tab from a fresh page load.

```
<button class="focus:outline-2 focus:outline-offset-2 focus:outline-blue-600">focus:</bu
<button class="focus-visible:outline-2 focus-visible:outline-offset-2 focus-visible:outl
<button class="outline-hidden">outline-hidden, no replacement</button>
<input class="border px-2 py-1 focus-visible:outline-2 focus-visible:outline-blue-600">
```

```
MOUSE click   :focus-visible   outline shown
  focus:           false          yes  (solid 2px)     the ring appears after every mou
  focus-visible:   false          no                   the ring stays away for mouse us
  outline-hidden   false          no
  input            true           yes  (solid 2px)     a text field always counts as ke
```

```
TAB key        :focus-visible   outline shown
  focus:           true             solid 2px
  focus-visible:   true             solid 2px
  outline-hidden   true             none                  <- a keyboard user cannot see wh
  input            true             solid 2px
```

The pattern to adopt: use `focus-visible:` for the ring (mouse users on buttons are not shown it, keyboard users always are), and never use `outline-hidden` or `outline-none` without putting a visible replacement on `focus-visible:`. The third button is the classic bug: it looks tidier and has no focus indicator at all. Chapter 8 explained that `outline-hidden` keeps a transparent outline in forced-colours mode; it still shows nothing in normal mode.

Text for Screen Readers Only

A star icon on a link has no text. You can add text that is invisible on screen but still read aloud. But `hidden` and `invisible` remove text from assistive technology as well. I asked Chrome for the accessibility tree and read the name it computed for each of five links:

```
<a href="#"><span class="sr-only">Profile (sr-only)</span><i>★</i></a>
<a href="#"><span class="hidden">Profile (hidden)</span><i>★</i></a>
<a href="#"><span class="invisible">Profile (invisible)</span><i>★</i></a>
<a href="#" aria-label="aria-label only"><i>★</i></a>
```

```
accessible name of each link, from the accessibility tree:
  sr-only      "Profile (sr-only) ★"
  hidden       "★"                      the text is gone
  invisible    "★"                      the text is gone
  aria-label   "aria-label only"

sr-only computed:  position absolute, width 1px, height 1px, overflow hidden, white-spac
```

Only `sr-only` keeps the words in the name. The same trick builds a *skip link*, a link that is invisible until a keyboard user reaches it:

```
<a href="#main" class="sr-only focus:not-sr-only focus:absolute focus:left-2 focus:top-2
```

```
before focus:  position absolute, width 1px, background transparent
after Tab to it:  position absolute, width 123.391px, background rgb(255,255,255), left
```

`focus:not-sr-only` undoes the hiding exactly when the link receives focus, which is why the skip link is both hidden for mouse users and reachable by keyboard.

Contrast

The WCAG minimum for normal text is 4.5 to 1 (3 to 1 for large text and interface parts). Tailwind's palette is not guaranteed to meet it, and the common mistakes are light greys and white text on mid-tone colours. I measured the ratio for common pairs by reading each element's computed colours, converting them to sRGB through a canvas, and applying the WCAG formula:

TEXT	BACKGROUND	RATIO	RESULT
`text-slate-400` (#90a1b9)	white	2.63	fails
`text-gray-400` (#99a1af)	white	2.60	fails
`text-slate-500` (#62748e)	white	4.76	passes
`text-gray-500` (#6a7282)	white	4.84	passes
`text-slate-500`	`bg-slate-100` (#f1f5f9)	4.35	fails for normal text
`text-slate-600` (#45556c)	white	7.58	passes
white	`bg-blue-500` (#2b7fff)	3.76	large text only
white	`bg-blue-600` (#155dfc)	5.25	passes
white	`bg-red-500` (#fb2c36)	3.81	large text only
white	`bg-emerald-500` (#00bc7d)	2.47	fails
white	`bg-emerald-700` (#007a55)	5.36	passes

TEXT	BACKGROUND	RATIO	RESULT
`text-slate-900`	`bg-yellow-300` (#ffdf20)	13.44	passes

Two surprises. A grey that passes on white (`slate-500`, 4.76) fails on a light grey panel (4.35). And white on `emerald-500` is only 2.47, worse than most people guess; one step darker passes. Users can also ask for more or less contrast, and Tailwind has variants for that. With `contrast-more:` and `contrast-less:` set on a paragraph and a border, I switched Chrome's emulated `prefers-contrast`:

```
<p class="text-slate-500 contrast-more:text-slate-900">...</p>
<div class="border border-slate-300 contrast-more:border-slate-900 contrast-less:border-
```

```
prefers-contrast: no-preference   text slate-500 (oklch 0.554)   border slate-300 (0.869
prefers-contrast: more            text slate-900 (oklch 0.208)   border slate-900 (0.208
prefers-contrast: less            text slate-500 (unchanged)     border slate-100 (0.968
```

Reduced Motion

Some people feel ill from animation. The setting is `prefers-reduced-motion`; Tailwind gives two variants, `motion-reduce:` (the user asked for less) and `motion-safe:` (they did not). I used four elements and switched the emulated preference:

```
<div class="animate-spin">                                    <!-- m1: ignores the prefere
<div class="animate-spin motion-reduce:animate-none">          <!-- m2 -->
<div class="motion-safe:animate-spin">                         <!-- m3 -->
<div class="transition duration-700 hover:translate-x-4 motion-reduce:transition-none">
```

```
                                 no preference      prefers-reduced-motion: reduce
m1  animate-spin                 spin               spin            <- still spinning
m2  motion-reduce:animate-none   spin               none
m3  motion-safe:animate-spin     spin               none
m4  transition-property          (all listed)       none            (duration 0.7s is
```

The lesson is in the first row: nothing is automatic. An animation without one of the two variants runs for everybody. The safer default is `motion-safe:` on the animation itself, so that it exists only when the user has not asked for less.

Forced Colours

In Windows High Contrast and similar modes the browser replaces your colours with the user's chosen system palette. The variant is `forced-colors:`, with `not-forced-colors:` for the opposite and `forced-color-adjust-none` to opt an element out. I switched Chrome's emulated `forced-colors: active`:

```
x1 <button class="bg-blue-600 text-white forced-colors:bg-[ButtonFace] forced-colors:tex
x2 <button class="bg-blue-600 text-white">                          (no override)
x3 <div class="bg-blue-600 text-white forced-color-adjust-none">
x4 <div class="not-forced-colors:bg-blue-600">
```

```
                normal                          forced-colors: active
x1  bg blue-600, text white          ->    background white, text black     (system col
x2  bg blue-600, text white          ->    background white, text black     (replaced a
x3  bg blue-600, text white          ->    unchanged: forced-color-adjust: none keeps y
x4  bg blue-600                      ->    transparent: the not-forced-colors: rule no
matchMedia('(forced-colors: active)').matches  ->  true
```

The browser already handled x2, so you rarely need to write `forced-colors:` yourself. The value of `forced-color-adjust-none` is also its danger: x3 kept your colours, so you now own the contrast. The cases that need a variant are decorative separators and custom controls built from background colours alone: in forced colours a background-only bar can vanish, so give it a border or an outline. I tested with Chrome's emulation, not a real Windows contrast theme.

Logical Properties and Right-to-Left

`ml-4` always means "left". For a right-to-left language (Arabic, Hebrew) you want "start". Tailwind's logical utilities (`ms-`, `me-`, `ps-`, `pe-`, `border-s`, `start-`, `end-`, `text-start`) follow the writing direction. I laid out one block with logical classes and one with physical ones, then set `dir="rtl"` on their container:

```
<div class="ms-4 me-8 ps-2 pe-6 border-s-4 text-start">logical</div>
<div class="ml-4 mr-8 pl-2 pr-6 border-l-4 text-left">physical</div>
<div class="rtl:bg-red-500 ltr:bg-green-500 rtl:ms-10">rtl/ltr</div>
<div class="relative h-8"><span class="absolute start-0">start-0</span><span class="abso
```

```
                          LTR                                          RTL (dir="rtl" on
logical   margin-left 16  margin-right 32  border-left 4      ->  margin-left 32  margin
physical  margin-left 16  margin-right 32  border-left 4      ->  unchanged: still left
rtl/ltr   green, margin-left 0                                ->  red, margin-right 40 (
start-0   span at x = 16, end-0 span at x = 826               ->  start-0 at x = 822, en
```

The logical block mirrored by itself; the physical one stayed put. `rtl:` and `ltr:` are for the cases logical utilities cannot express (an icon that must flip, or a different colour). Tailwind 4.3.3 also has logical *block*-axis utilities. I built each and checked the generated declaration:

CLASS	GENERATES	EXISTS IN 4.3.3
`pbs-4`, `pbe-1`	`padding-block-start`, `padding-block-end`	yes
`mbs-3`, `mbe-2`	`margin-block-start`, `margin-block-end`	yes
`inline-4`, `block-8`	`inline-size`, `block-size`	yes
`min-inline-2`, `max-block-10`	`min-inline-size`, `max-block-size`	yes
`border-bs-2`	`border-block-start-width`	yes
`inset-s-1`, `inset-bs-0`, `start-0`	`inset-inline-start`, `inset-block-start`	yes
`pis-2`, `pie-3`, `mis-1`, `mie-4`	nothing	**no**: use `ps-`, `pe-`, `ms-`, `me-`

Note the last row: the CSS property names (`padding-inline-start`) suggest `pis-`, but Tailwind's inline-axis names are the shorter `ps-` and `pe-`, and the block-axis ones are `pbs-` and `pbe-`. I did not check which Tailwind release added the block-axis utilities; they exist in 4.3.3.

Print Styles

The `print:` variant applies under `@media print`. I emulated print media for the computed styles, and used Chrome's PDF output (A4, 0.4 inch margins) to see real page breaks:

```
<nav class="bg-slate-200 print:hidden">...</nav>
<div class="hidden print:block">only in print</div>
<div class="bg-yellow-200 print:bg-white print:text-black">print colours</div>
```

```
                              screen                print media
nav (print:hidden)            display block         display none
"only in print"               display none          display block
bg-yellow-200 print:bg-white  yellow                rgb(255,255,255), text rgb(0,0,0)
```

The PDF test used a navigation bar (160 px tall) and three sections of 320 px:

```
a.html   nav, three sections                                          2 PDF page
b.html   same, nav has print:hidden                                   1 PDF page
c.html   b.html + break-before-page on sections two and three         3 PDF pages
```

Hiding the navigation brought the document from two pages to one; `break-before-page` forces a section onto its own page. `break-inside-avoid` is the companion that keeps a card from splitting across two pages. One thing I assumed and got wrong: Tailwind 4.3.3 has no `print-color-adjust` utility (`print-color-adjust-exact` produced no rule). The arbitrary property `[print-color-adjust:exact]` does work, and does generate `print-color-adjust: exact`. Whether backgrounds print also depends on the user's print dialog; my PDF test asked for them explicitly (`printBackground`). The site's own PDF builder works by loading chapters in headless Chrome, so `print:` rules in a chapter should apply to its PDF (I did not test that builder).

An Accessibility Review Checklist for Tailwind

- Every interactive element has a visible keyboard focus style on `focus-visible:`; no bare `outline-hidden` or `outline-none`.

- Icon-only controls have a name: `sr-only` text or `aria-label`; never `hidden` or `invisible` text.
- A skip link exists (`sr-only focus:not-sr-only`).
- Text and interface colours pass the WCAG ratios (measure; do not guess: `emerald-500` with white text fails); `contrast-more:` variants for subtle greys.
- Every animation is under `motion-safe:` or has `motion-reduce:` alternatives.
- No meaning conveyed by background colour alone (forced colours removes it); `forced-color-adjust-none` only where you have checked contrast.
- Layout uses logical utilities (`ms-`, `ps-`, `start-`, `text-start`); `rtl:` only for things that must flip.
- Print: hide navigation, show URLs or notes if needed, set break behaviour, and test the PDF.

MISTAKES WHILE PREPARING THIS CHAPTER

1. My first contrast table said 21 to 1 for most pairs. Those classes were not in the built CSS (the scanner had only seen the page), so the text was black on white. **How to avoid it:** a result of exactly 21.00 (or 1.00) for a colour pair is a red flag; build the classes and add a check you can predict.

2. I counted four Tab presses to reach the skip link and landed on the wrong control. The page has four earlier focusable elements. **How to avoid it:** print the focused element's id after each key press instead of counting.

3. I assumed a `print-color-adjust-exact` utility. It produced no rule. **How to avoid it:** build the class and look for a rule before writing it into a lesson.

4. My first print test fitted on one page whether or not the navigation was hidden, so it proved nothing. I made the sections taller. **How to avoid it:** work out what result would show a difference before running the test.

Hands-On Exercises

EXERCISE 1

Make four controls (`focus:`, `focus-visible:`, `outline-hidden` alone, an input). Click each with the mouse and Tab through them, and report `:focus-visible` and the outline for each case. Then build five links (`sr-only`, `hidden`, `invisible`, `aria-label`, a skip link) and read their names from the accessibility tree.

View solution

EXERCISE 2

Measure the WCAG contrast ratio of twelve Tailwind colour pairs by reading computed colours. Include a result you can predict as a check. Then show what `contrast-more:`, `motion-reduce:`, `motion-safe:` and `forced-colors:` change by switching the emulated media features.

View solution

EXERCISE 3

Build a block with logical classes and one with physical classes, switch the container to `dir="rtl"` and report what moved. Then make a page that prints to one PDF page with `print:hidden` on the navigation and to three pages with `break-before-page`, and report the page counts.

View solution

CHAPTER 9 QUICK REFERENCE

- Focus ring on `focus-visible:`: mouse clicks on buttons do not show it, keyboard always does. `outline-hidden` with no replacement left keyboard focus invisible
- `sr-only` keeps text in the accessible name (the accessibility tree showed it); `hidden` and `invisible` remove it. Skip link: `sr-only focus:not-sr-only focus:absolute focus:left-2 focus:top-2 focus:bg-white`
- Contrast is not guaranteed by the palette: slate-400/white 2.63, white on emerald-500 2.47 fail; slate-500/white 4.76 and white on blue-600 5.25 pass. `contrast-more:` / `contrast-less:` respond to `prefers-contrast`
- Animations run for everyone unless you add `motion-safe:` or `motion-reduce:`; `motion-reduce:transition-none` sets the property to none
- `forced-colors:` / `not-forced-colors:` / `forced-color-adjust-none`: system colours replace yours; backgrounds alone can vanish; opting out makes contrast your job (tested with emulation only)
- Logical utilities mirror in RTL by themselves (`ms-` `me-` `ps-` `pe-` `border-s` `start-` `end-` `text-start`); physical `ml-` does not. `pbs-` / `pbe-` / `mbs-` / `mbe-` / `inline-` / `block-` exist in 4.3.3; `pis-` / `mis-` do not

- `print:hidden` took a test document from 2 PDF pages to 1; `break-before-page` gave 3; no `print-color-adjust-*` utility (use `[print-color-adjust:exact]`)

COMING NEXT

Chapter 10 is the capstone: a small themed, accessible component kit (button, input, card, tabs, dialog) with design tokens, two brand themes plus dark mode, `cva` variants and a documentation page, wired into Vite and into a second tool, with a shipping checklist and the course recap. Start with *Tailwind CSS Intermediate/Advanced 10*.

Tailwind CSS Intermediate/Advanced

Chapter 10 · Capstone: A Themed Component Kit

This chapter uses everything in the course on one small project: a component kit with a button, an input, a card, tabs and a dialog; design tokens with two brand themes and dark mode; `cva` variants; a documentation page; and the same kit running inside two different tools. As in the earlier chapters, nothing is described that I did not build and measure, and what went wrong on the way is listed at the end.

> **EVERY EXAMPLE HERE WAS RUN, EXCEPT WHERE MARKED**
>
> Tailwind 4.3.3, Vite 8.3.4, React 19.3.0, Astro 7.3.8 with @astrojs/react 7.0.1, @radix-ui/react-tabs 1.2.0 and react-dialog 1.3.0, class-variance-authority 0.7.1, clsx 2.1.1, tailwind-merge 3.7.0, TypeScript 6.0.3 (the version npm installed), prettier 3.9.10 with prettier-plugin-tailwindcss 0.8.1, eslint 10.12.0 with eslint-plugin-better-tailwindcss 4.9.0, Node 24.18.0, Windows 11, headless Chrome 154 driven over the DevTools Protocol. **Not run:** a real screen reader, a real Windows contrast theme, Firefox and Safari, the Next.js wiring (the outline allowed Astro or Next.js; I chose Astro, the default), a deployed site, and any test of the kit's keyboard behaviour beyond the ones listed below.

The Plan

The kit has three layers, each from an earlier chapter: **tokens** (chapter 2: semantic variables that the themes change), **components** (chapter 6: `cva` for variants, `cn` to merge, Radix for behaviour) and **checks** (chapters 7 and 9: formatting, linting, contrast, keyboard and motion tests). The folder:

```
kit/
├── index.html            no-flash theme script (chapter 5)
├── vite.config.ts
├── eslint.config.js      better-tailwindcss, entry point = tokens.css
├── .prettierrc           tailwind plugin, stylesheet, cn and cva
├── .gitignore            node_modules and dist  (this matters, see below)
└── src/
    ├── ui/
    │   ├── tokens.css    themes and the @theme mapping
    │   ├── cn.ts         clsx + tailwind-merge
    │   ├── button.tsx    input.tsx   card.tsx   tabs.tsx   dialog.tsx
```

```
    └── docs/
        ├── App.tsx       the documentation page (exports App)
        └── main.tsx      mounts App for Vite
```

Tokens and Themes

The tokens are plain CSS variables on `:root`, replaced by attribute selectors for dark mode and for each brand. A separate `@theme inline` block turns them into Tailwind colour names (`bg-primary`, `text-muted`), exactly the pattern from chapter 2: `inline` is what lets the utilities follow whichever variable value is active. The `dark:` variant is redefined to follow the `data-mode` attribute instead of the operating system setting.

```
@import 'tailwindcss';

/* The kit's own files, wherever the app that imports this stylesheet lives */
@source '../';

/* Dark mode follows the data-mode attribute, not the OS setting */
@custom-variant dark (&:where([data-mode="dark"], [data-mode="dark"] *));

/* Semantic tokens are plain variables, switched by data-brand and data-mode */
:root {
  color-scheme: light;
  --surface: oklch(99% 0.003 250);
  --surface-raised: oklch(100% 0 0);
  --ink: oklch(22% 0.03 255);
  --ink-muted: oklch(45% 0.03 255);
  --line: oklch(86% 0.01 255);
  --primary: oklch(50% 0.2 255);
  --primary-ink: oklch(99% 0 0);
  --danger: oklch(50% 0.2 27);
  --danger-ink: oklch(99% 0 0);
  --ring: oklch(55% 0.2 255);
}
[data-mode='dark'] {
  color-scheme: dark;
  --surface: oklch(18% 0.02 255);
  --surface-raised: oklch(23% 0.025 255);
  --ink: oklch(95% 0.01 255);
  --ink-muted: oklch(75% 0.02 255);
  --line: oklch(36% 0.025 255);
```

```
    --primary: oklch(72% 0.14 255);
    --primary-ink: oklch(18% 0.03 255);
    --danger: oklch(72% 0.15 27);
    --danger-ink: oklch(18% 0.03 27);
    --ring: oklch(75% 0.14 255);
  }
  [data-brand='sunset'] {
    --primary: oklch(52% 0.18 40);
    --ring: oklch(58% 0.18 40);
  }
  [data-brand='sunset'][data-mode='dark'] {
    --primary: oklch(75% 0.15 55);
    --primary-ink: oklch(20% 0.03 55);
    --ring: oklch(78% 0.15 55);
  }

  @theme inline {
    --color-surface: var(--surface);
    --color-raised: var(--surface-raised);
    --color-ink: var(--ink);
    --color-muted: var(--ink-muted);
    --color-line: var(--line);
    --color-primary: var(--primary);
    --color-primary-ink: var(--primary-ink);
    --color-danger: var(--danger);
    --color-danger-ink: var(--danger-ink);
    --color-ring: var(--ring);
    --radius-control: 0.5rem;
    --radius-panel: 0.75rem;
  }

  @layer base {
    body {
      background: var(--surface);
      color: var(--ink);
    }
  }
```

The brand and mode are set by two attributes on `<html>`. `@source '../';` at the top points the scanner at the kit's own files, wherever the app that imports this stylesheet lives; I did not have that line at first (see the Astro section for what happened).

The Components

Button uses `cva` for four intents and three sizes, merges the caller's `className` with `cn`, shows its focus ring only on `focus-visible:` (chapter 9), and turns off its colour transition under `motion-reduce:`.

```
import { cva, type VariantProps } from 'class-variance-authority';
import type { ButtonHTMLAttributes } from 'react';
import { cn } from './cn';

export const buttonStyles = cva(
  'inline-flex items-center justify-center gap-2 rounded-control font-medium transition-
  {
    variants: {
      intent: {
        primary: 'bg-primary text-primary-ink hover:bg-primary/90',
        secondary: 'border border-line bg-raised text-ink hover:bg-line/40',
        ghost: 'text-ink hover:bg-line/40',
        danger: 'bg-danger text-danger-ink hover:bg-danger/90',
      },
      size: { sm: 'h-8 px-3 text-sm', md: 'h-10 px-4 text-sm', lg: 'h-12 px-6 text-base'
    },
    defaultVariants: { intent: 'primary', size: 'md' },
  },
);

type Props = ButtonHTMLAttributes<HTMLButtonElement> & VariantProps<typeof buttonStyles>

export function Button({ intent, size, className, type = 'button', ...rest }: Props) {
  return <button type={type} className={cn(buttonStyles({ intent, size }), className)} {
}
```

Input is a labelled field. The label is tied to the input by `htmlFor`, and when there is an error the input gets `aria-invalid` and `aria-describedby` pointing at the message; the red border comes from the `aria-invalid:` variant, not a separate class, so styling and accessibility cannot drift apart (chapter 4). **Tabs** and **Dialog** wrap Radix, which supplies the behaviour, and add only classes. The tab styling hangs on `data-[state=active]:` (chapter 6); the dialog fades in with `starting:opacity-0` and turns that off under `motion-reduce:`:

```
import * as D from '@radix-ui/react-dialog';
import type { ReactNode } from 'react';

export const Dialog = D.Root;
export const DialogTrigger = D.Trigger;
export const DialogClose = D.Close;

export function DialogContent({ title, description, children }: { title: string; descrip
  return (
    <D.Portal>
      <D.Overlay className="fixed inset-0 bg-ink/50 transition-opacity duration-150 moti
      <D.Content className="fixed top-1/2 left-1/2 w-[min(28rem,calc(100vw-2rem))] -tran
        <D.Title className="text-lg font-semibold">{title}</D.Title>
        <D.Description className="mt-1 text-sm text-muted">{description}</D.Description>
        <div className="mt-4">{children}</div>
      </D.Content>
    </D.Portal>
  );
}
```

The documentation page (`App.tsx`) shows every component in every variant, with buttons that set the brand and the mode and save them in `localStorage`. The theme is applied before the page paints by a small classic script in the `<head>` (chapter 5), so a saved dark choice does not flash light first.

The Checks I Ran

1. Types, formatting, linting

```
"check": "tsc --noEmit && prettier --check src && eslint src"
```

```
npm run check           exit 0   (tsc: no errors; "All matched files use Prettier code 

first run, before formatting: prettier reported 5 files; eslint reported 6 warnings, amo
  "-translate-x-1/2, -translate-y-1/2" can be simplified to "-translate-1/2"
  "focus-visible:outline-offset-[-2px]" can be simplified to "focus-visible:-outline-off
  Incorrect class order (starting:opacity-0 placed before motion-reduce:transition-none)
I applied prettier --write and eslint --fix and re-ran the browser tests: results unchan
```

```
seeded errors (bg-raised changed to bg-raisd, and p-3 added next to p-5):
  error  Unknown class detected: bg-raisd
  error  Conflicting class detected: "p-3" and "p-5" apply the same CSS properties: "pad
  (the && chain stopped at Prettier first, because the typo also unsorted the list; ESLi
```

2. Contrast, for every brand in every mode

Using the method from chapter 9 (computed colour, converted to sRGB through a canvas, WCAG formula), I switched the page through the four theme combinations with real clicks and measured nine text pairs and the focus ring. The last row is a *control*: ink text on the primary colour, which must fail, to prove the method can report a failure.

PAIR	OCEAN LIGHT	OCEAN DARK	SUNSET LIGHT	SUNSET DARK
body text on page	16.86	16.26	16.86	16.26
muted text on card	7.46	7.58	7.46	7.58
primary button	5.87	7.58	5.83	7.81
danger button	6.47	7.18	6.47	7.18
secondary button	17.34	14.54	17.34	14.54
input label	16.86	16.26	16.86	16.26
placeholder	7.46	7.58	7.46	7.58
inactive tab	7.26	8.48	7.26	8.48
error text	6.45	7.13	6.45	7.13
focus ring against page (needs 3)	4.76	8.41	4.51	9.00
CONTROL: ink on primary	2.88 (fails)	2.15 (fails)	2.90 (fails)	2.01 (fails)

Every real pair passes 4.5 (the ring needs only 3). The numbers are not all alike, so the measurement is not stuck, and the control fails in all four. I chose the colours with those targets in mind and they passed on the first run; that will not usually happen, and it is the reason to keep the test. Hover colours (the 90% mix on buttons) and the disabled state were not measured.

3. Keyboard, dialog, motion and persistence

```
TAB ORDER: brand-ocean > brand-sunset > mode-light > mode-dark > b-primary > b-secondary
           > email > name > tab-one > tab panel > open-dialog > (leaves the page) > back
           (the disabled button is skipped; tab-two and tab-three are NOT in the Tab ord
FOCUS RING on the first control: :focus-visible true, outline solid 2px in the ring colo
TABS: focus on tab-one, ArrowRight -> tab-two selected and focused, panel text "Second p
DIALOG open: role dialog, aria-labelledby and aria-describedby set, focus moved inside (
             Tab pressed 3 times: focus stays inside (Cancel > Delete > ...); Escape: di
INPUT: after typing "nope": aria-invalid=true, aria-describedby=email-error, message "En
REDUCED MOTION: button transition-property = none (against "color, background-color, bor
DIALOG animation: opacity 0 at insertion, 1 after 400 ms
PERSISTENCE: choose sunset + dark, reload -> data-brand/data-mode = sunset/dark, localSt
             body background over the first 200 ms of frames: one value only (dark), no
```

The Second Wiring: Astro

The same components, unchanged, rendered inside an Astro page as a React island (`<App client:load />`), importing the kit's files from the folder next to it. The Astro configuration is the one from chapter 5: the React integration and `@tailwindcss/vite` inside `vite`. The no-flash script is `is:inline` (chapter 5 again, or Astro turns it into a deferred module).

```
// astro/astro.config.mjs
export default defineConfig({
  integrations: [react()],
  vite: { plugins: [tailwindcss()] },
});

// astro/src/pages/index.astro  (frontmatter)
import { App } from '../../../kit/src/docs/App';
// body:  <App client:load />
```

```
first build:   exit 0, no warning.   CSS 6,396 B.   bg-primary: 0 matches. outline-ring:
               The page rendered unstyled: the kit's classes were outside the Astro proj

fix:           one line at the top of the SHARED stylesheet (kit/src/ui/tokens.css):   @
second build:  exit 0.   CSS 15,092 B.   bg-primary: 3. outline-ring: 1. rounded-control
```

This is the monorepo failure from chapter 5, met again in a project I had designed to avoid it: no error, an almost empty stylesheet. Putting the `@source` line in the shared stylesheet, relative to

that file, means every app that imports the kit gets the right sources without its own configuration. With that, I ran the same keyboard, contrast, dialog and persistence scripts against the Astro build: **every result was identical** to the Vite build. Then I took screenshots (900 by 1100) of both builds in all four theme combinations and compared them:

```
                  same build, second run      Vite against Astro
  ocean-light     0 of 990,000 pixels         0 of 990,000
  ocean-dark      0                           0
  sunset-light    0                           0
  sunset-dark     0                           0
  ocean-light against ocean-dark: 981,557 pixels differ   ocean-light against sunset-light
```

The two controls matter: the same build twice gives zero (so the comparison is not noisy) and the different themes give large differences (so it can see a change). Both wirings produce the same pixels. Sizes: the Vite build is 14.06 kB of CSS (3.64 kB gzipped) and 308.5 kB of JavaScript (96.9 kB gzipped); the Astro build has 15.1 kB of CSS and loads the same components as an island.

Two Traps I Hit on the Way

The build folder is scanned. After formatting, the CSS suddenly measured 29.4 kB instead of 14.1 kB. My first guess was that the test scripts I had put in the project folder were being read (they contain many class-like words). I moved them out and the size stayed at 28.8 kB, so the guess was wrong. The real cause was `dist/`: the first build had no output folder; the second build scanned the first build's 308 kB JavaScript bundle and generated a rule for every class-like word in it.

```
  no .gitignore:     build 1 -> 14.06 kB     build 2 (dist/ now exists) -> 28.82 kB
  .gitignore with "node_modules" and "dist":  build 1 -> 14.06 kB     build 2 -> 14.06 kB
```

That is the `.gitignore` behaviour from chapter 1 working in your favour: always keep `dist` and `node_modules` in it, or the output grows by itself. And the second trap is the one above: a shared stylesheet outside the app needs its own `@source`.

Checklist: Before You Ship a Kit

- **Sources.** The shared stylesheet has `@source` for its own files; the build output folder is ignored; a second build gives the same CSS size as the first.
- **Tokens.** Components use semantic names only (`bg-primary`), never palette names; every brand and mode defines every token.
- **Contrast.** A script measures every text pair in every theme and includes a control that must fail.
- **Keyboard.** Tab order, arrow keys in tabs, dialog focus in and out, a visible `focus-visible:` ring on everything interactive.
- **Preferences.** `motion-reduce:` on every transition and animation; saved theme applied before paint.
- **Types and lint.** `tsc` , Prettier and ESLint pass in one command; a seeded typo is caught.
- **Each wiring.** The same tests and a pixel comparison per tool; do not assume the second one works because the first does.
- **Browsers.** Chrome 111, Safari 16.4, Firefox 128 are the minimum for Tailwind 4 (chapter 7); I tested Chrome only.
- **Not done here, but before shipping:** a real screen-reader pass, Firefox and Safari, a real forced-colours theme, and a release process (versioning, a changelog).

MISTAKES WHILE PREPARING THIS CHAPTER

1. I blamed the wrong cause for the CSS doubling. I assumed my test scripts were being scanned; moving them changed nothing. **How to avoid it:** change one thing at a time and re-measure; here, deleting `dist/` and building twice found the cause in two builds.

2. The Astro page built, exit 0, with no kit styles. I had not carried the chapter 5 monorepo lesson into my own design. **How to avoid it:** after wiring a second tool, grep its CSS for a kit class before looking at anything else.

3. My theme-persistence test seemed to show the feature was broken. The page had scrolled, so my script clicked at a position that was not on screen. **How to avoid it:** scroll the target into view, or check that the click changed something, before drawing conclusions about the feature.

4. My single `check` command stopped at the first failing tool. The ESLint report for the seeded typo never appeared until I ran it directly. **How to avoid it:** run each tool once on its own after changing the chain, or run them with `;` or a task runner that reports all of them.

5. I tried to create the whole project with one long shell here-document and the shell rejected it, writing nothing. How to avoid it: write files with the editor tool, and confirm the files exist before the next step.

Hands-On Exercises

EXERCISE 1

Build the tokens (two brands, light and dark), a `cva` button and an input with error state, then measure every text pair's contrast in all four theme combinations with a control that must fail. Fix anything that fails.

View solution

EXERCISE 2

Render the same components in a second tool (Astro, or Next.js if you prefer). Show the first build's CSS lacks the kit's classes, fix it with one `@source` line in the shared stylesheet, and prove the two builds match with screenshots, including a same-build control and a different-theme control.

View solution

EXERCISE 3

Write one `check` command for types, formatting and linting; seed a typo and a conflict and show the report. Then show that a project without a `.gitignore` produces a bigger stylesheet on its second build, and fix it.

View solution

CHAPTER 10 QUICK REFERENCE

- Layers: tokens (CSS variables + `@theme inline`), components (`cva` + `cn` + a headless library), checks (types, Prettier, ESLint, contrast, keyboard, pixels)

- Themes by attribute: `data-brand` and `data-mode` on `<html>`, a classic head script before paint, `@custom-variant dark` following `data-mode`
- Measured: all nine text pairs pass in four themes (lowest 5.83), the control fails (2.01 to 2.90); Vite CSS 14.06 kB; Astro CSS 15.1 kB; Vite and Astro screenshots identical (0 of 990,000 pixels)
- Shared kit in another app: `@source '../';` inside the kit's own stylesheet; without it the Astro build exited 0 with 6,396 B of CSS and no kit classes
- Keep `dist` and `node_modules` in `.gitignore`: without it the second build's CSS doubled (14.06 to 28.82 kB)
- Every test needs a control: the same page twice (should match), a different theme (should differ), a colour pair that must fail

Course Complete: What You Can Now Do

You have worked through the ten chapters of Tailwind CSS Intermediate/Advanced. Each one is a skill, and each was checked in a real build rather than taken from memory:

CHAPTER	YOU CAN NOW	CHAPTER'S KEY MEASURED RESULT
1 Engine	explain how v4 finds classes, use `@source`, read the layered output	dynamic class names fail; `.gitignore` is honoured even without git
2 Design Tokens	build multi-theme systems with `@theme inline` and variables	inline tokens are not written to the CSS; themes switch by attribute
3 Utilities and Plugins	write `@utility`, `@custom-variant` and small plugins	a bad `--value()` prints an error but exits 0
4 Advanced Variants	use groups, peers, `has-`, `not-`, `in-`, `*:`, data and aria variants	`in-hover:` matches whenever the pointer is on the page
5 Frameworks	wire Tailwind into seven toolchains	a late theme script flashed the wrong colour for 175 ms
6 Component Patterns	merge classes, build typed variants, choose a library	class order in the attribute never decides conflicts
7 Tooling	size, time and lint a project	size follows distinct class-and-variant combinations
8 Migration	migrate v3 to v4 and verify it	the upgrade tool left 4,279 differing pixels; by hand, 0

CHAPTER	YOU CAN NOW	CHAPTER'S KEY MEASURED RESULT
9 Accessibility	handle focus, contrast, motion, forced colours, RTL and print	`outline-hidden` alone left keyboard focus invisible
10 Capstone	ship a themed, checked kit in two tools	identical pixels in Vite and Astro; the build folder doubled the CSS

Where to go next: the prerequisite course *Tailwind CSS Fundamentals* if any chapter felt thin; *Web Accessibility (a11y)* for the standards behind chapter 9 and a real screen-reader pass for your own kit; *CSS Frameworks* 7 on shadcn/ui if you want to start from an existing kit; and your own project: take the checklist above and run it on something you have already built.