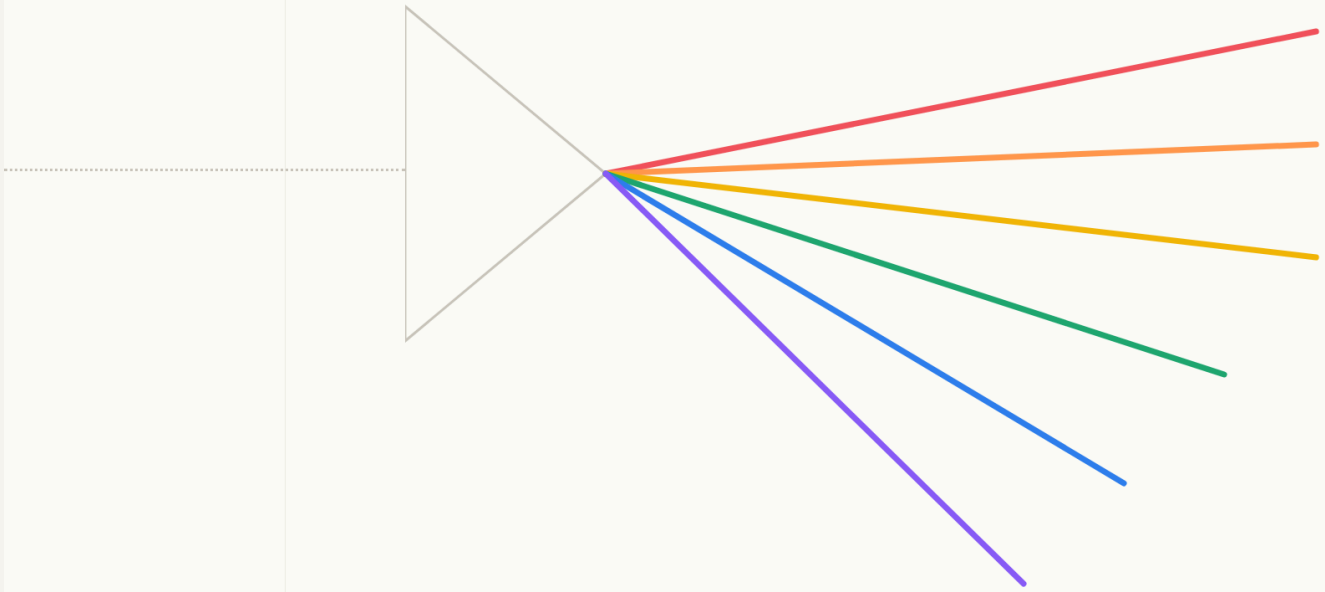


Web Components & Design Systems

Built for Humans, Engineered to Be Agent-Ready

Sandeep Kumar Patel

A TECHNICAL BOOK

Web Components & Design Systems

Built for Humans, Engineered to Be Agent-Ready

Sandeep Kumar Patel

August 2026

COPYRIGHT

Contents

Preface

This book grew out of a specific annoyance: I kept reading advice about web components that had never actually been run.

Not maliciously — most of it was written in good faith, by people repeating what they'd heard rather than what they'd verified. "Closed shadow roots are more secure." "You can't style slotted content." "`attributeChangedCallback` fires once per change." Every one of those is false, in a specific, checkable way, and every one of them is still out there in blog posts and Stack Overflow answers with dozens of upvotes. The platform moved fast enough, and the folklore calcified early enough, that a lot of what's "commonly known" about web components describes a browser that stopped existing around 2020.

So the discipline behind this book is simple to state and was not simple to hold to for twenty-seven chapters: **if it's in here, I ran it.** Not "the spec allows it" — ran it, in a real browser, and printed what actually happened. Where I was wrong on a first attempt, that's in here too, because the corrections are usually more instructive than the parts I got right the first time. Chapter 17's `:host(:dir(rtl))` rule is the clearest example — a rule I taught, then rigorously re-tested months later, then removed, because it turned out never to have been necessary at all. That kind of reversal doesn't happen if you never check.

Who this is for. Front-end developers, UI engineers, and design-system teams who want to build reusable UI on the platform itself rather than on top of one framework's release schedule — and who want the reasoning shown, not just the conclusion. You don't need prior web-components experience; Part I starts from the platform's own primitives. You do need to already be comfortable with JavaScript and the DOM.

What you'll build. Rather than a dozen disconnected snippets, this book has one running project: **Prism DS**, a small but genuinely real design system, built up one chapter at a time — tokens, themed components, a manifest, a documentation pipeline, and eventually an MCP server that lets an AI agent consume it correctly. Every mechanism the platform offers gets demonstrated against this same accumulating codebase, so by Part IV you're not learning `attachInternals()` in the abstract — you're using it to fix a real bug in a component you already built.

Why the second half exists. Parts I through V would have been a complete, honest book on their own. I kept it going into Part VI because something changed while I was writing the rest of it: AI agents started being a second real consumer of design systems, alongside human developers, and the honest answer to "does this book's advice still apply" turned out to be yes, more than I expected — a design system's precision is exactly what makes it legible to an agent, for the same underlying reasons it makes a good API for a human. Part VI is where that argument gets made directly, with the same insistence on running the code rather than describing it.

How to read it. Cover to cover works, and the Prism DS case study is built to reward that. If you

already know custom elements, Chapter 3 is a fast refresher and you can pick up the design-system material from there. If you're starting a design system right now and don't have time for platform fundamentals first, start at Chapter 1, jump to Part III, then use Part IV as a template and the rest as reference. The Suggested Reading Paths page right after this one lays out a few concrete routes.

One last thing worth saying plainly, because the book itself argues for saying things plainly: this is a snapshot of a platform that is still moving. Every date, every browser-version claim, and every "as of this writing" in these pages was true in August 2026. Some of it won't be true by the time you're reading it — Firefox may have shipped scoped registries, a newer CSS feature may have closed a gap this book calls out as open. Where that's happened, trust the platform over the page, and treat the book's own discipline — check it yourself before you build on it — as the thing worth keeping, longer than any individual fact in here will stay current.

Prerequisites

This book assumes you're already comfortable writing JavaScript — not an expert, but past the beginner stage. Concretely, before Chapter 1, you should already be at ease with:

- **HTML and CSS**, including selectors, the box model, and Flexbox or Grid.
- **JavaScript classes** — `class Foo extends Bar`, `constructor()`, `super()`, instance methods, `static` members, and private fields (`#name`).
- **Arrow functions and destructuring** — `const { a, b } = obj`, `(x) => x * 2`.
- **Optional chaining and nullish coalescing** — `el?.value`, `a ?? b`.
- **Promises and `async` / `await`** — awaiting a `fetch()` call, chaining `.then()`, handling a rejected promise.
- **ES modules** — `import` / `export`, and the difference between a default and a named export.
- **Basic TypeScript type annotations** — reading `function f(x: string): number` well enough to follow along, even if you don't write TypeScript day to day.

If several of those are shaky rather than absent, that's fine — the **"Modern JavaScript & TypeScript Syntax Reference"** in Appendix A is a one-page refresher you can jump to the first time an unfamiliar piece of syntax shows up, then come straight back to where you left off. If most of them are genuinely new to you, this book will be a rough start; a JavaScript-fundamentals course or MDN's own JavaScript guide is a better place to begin, and this book will still be here once classes and `async` / `await` feel ordinary rather than new.

Nothing here is about intelligence or how "advanced" a reader you are — it's just the honest floor this book is written from, stated once up front instead of assumed silently and discovered the hard way in Chapter 2.

Conventions Used in This Book

A few recurring devices show up throughout — here's what each one means the first time you meet it, so you don't have to rediscover it mid-chapter.

Section references (`§N.M`). A reference like `§9.3` points to Chapter 9, section 3. These are used constantly, because almost nothing in this book stands alone — a technique introduced in one chapter is usually paid off, complicated, or corrected somewhere later, and the reference is how you find that thread without a full re-explanation each time.

Golden-rule callouts. A boxed, bolded one-sentence statement, marked with 🔑, appears wherever a section states a rule you should not break:

> 🔑 **Golden rule:** an example, stated exactly once, is the sentence worth remembering when everything around it is forgotten.

It always repeats a claim already made in the surrounding prose — that's intentional. It exists so that skimming back through a chapter later, the one sentence that matters is easy to find without rereading the paragraph around it.

"Why it matters" callouts. A shorter, unboxed-but-indented note marked with 💡 follows certain explanations — usually a named analogy — and translates the point back into the system's own technical terms, connecting the mental model to the actual code or decision it's grounding.

Named analogies. Roughly a dozen concepts in this book get a real-world comparison of their own — a shipping label, a dishwasher, a game of phone tag — reached for specifically where a concept has enough internal structure that a fourth restating of the abstract version stops helping. Each chapter that has one or more lists them again, briefly, in an "Analogies in this chapter" recap near the end, for quick review without re-reading the full explanation.

"What to remember." Every chapter closes with a short bulleted list, each bullet opening with a bold clause that stands on its own as a true statement — read just the bold parts of a chapter's closing list, and you have its actual argument compressed to a few lines.

Broken-code warnings. Where a code listing is deliberately shown broken — to demonstrate a bug before showing the fix — it's marked plainly right above the block, with a ⚠️ and a bold lead-in naming what's wrong with it:

> ⚠️ **Broken — do not copy.**

The exact wording varies chapter to chapter — "Fragile," "Wrong tool," a specific failure named outright

— but the ⚠️ and the bold lead-in are consistent throughout. Nothing in this book shows broken code without saying so first.

Figures. Three different kinds of figure appear, and they carry different weight. A **Mermaid diagram** shows a sequence or flow — timing, message-passing, a before/after. A **designed diagram graphic** is a styled illustration built for a comparison or taxonomy that doesn't have a natural sequence to walk through. A **real screenshot**, always captioned with the exact markup that produced it, is a verification artifact — proof that what the prose claims actually rendered.

Prism DS. The running example. Nearly every mechanism in this book gets demonstrated against the same accumulating design system rather than a disconnected snippet, so that by the time it shows up in Part IV's case study, its components, its base class, and its conventions are already familiar rather than introduced cold.

Code that says "I ran this." When the prose says a command was run, a value was measured, or a test passed, that claim was checked against real, working code before it was printed — not inferred from the specification or from memory. Where that discipline caught the author being wrong, the correction is left in rather than quietly edited away.

Suggested Reading Paths

This book is built to reward reading cover to cover — the Prism DS case study accumulates chapter by chapter, and later parts lean on components and decisions made earlier. But it doesn't demand that. A few concrete starting points, depending on what you actually came here for:

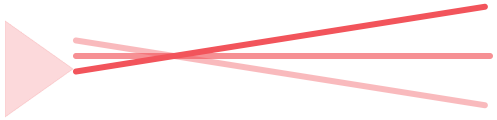
"I just want to learn web components." Parts I–II, then Chapter 20. That's the platform's four primitives, building real components by hand, and how to test what you built — a complete, self-contained arc without the design-system material.

"I already know custom elements." Start at Chapter 3 as a refresher on Shadow DOM's sharper edges, then jump to Chapter 12 for the vanilla capstone, then continue into Part III onward for the design-system material.

"I'm starting a design system." Chapter 1 for the case, then Part III for the fundamentals, then Part IV as a template for your own build — reference Part V (testing, packaging, adoption) as those questions actually come up, rather than reading it straight through first.

"I have a design system; I want to make it agent-ready." Start at §21.2 (the manifest, if you haven't built one yet), then read Part VI straight through, referencing Appendix C's scorecard and templates as you go.

Cover to cover. The default path, and the one the book is actually written for — Prism DS accumulates one decision at a time, and later chapters assume you've seen the earlier ones rather than re-explaining them.



PART I

Foundations of Web Components

The four platform primitives — Custom Elements, Shadow DOM, Templates & Slots, ES Modules — and why they hold up as a foundation.

Chapter 1: Why Web Components?

This chapter covers

- Why frameworks keep getting rewritten, and components don't have to be
- What "web components" actually means: four platform pillars
- Where the model fits best
- The stronger case for design systems specifically
- The honest limits of this approach
- Today's browser support, plainly

Introduction

If you've worked on front-end teams for more than a few years, you've probably lived through at least one framework migration you didn't choose and didn't especially want. I want to open with a concrete version of that experience. It's the reason this whole book exists, and naming it plainly up front will save us both a lot of hand-waving later.

The front-end industry has now built the same date picker at least five times.

The first was a jQuery plugin, around 2009: `$('.when').datepicker(options)`, a CSS file pasted into the project, and a silent prayer that no other plugin on the page also wanted to own `.calendar`. The second was a Backbone view. The third was an AngularJS directive. If you were there, you remember the feeling that this one, finally, was the permanent one. The fourth was a React class component. The fifth was the fourth again, rewritten with hooks, because by 2019 class components had started to read the way sideburns look in old photographs.

Here is the thing about those five date pickers: **the date picker never changed.** Same calendar grid. Same keyboard shortcuts. Same "weeks start on Monday in Europe" logic. Every rewrite was also a fresh chance to reintroduce the week-numbering bug somebody had fixed in 2011, faithfully, like a family recipe. What changed each time was the *wrapper*: the framework-shaped adapter between a perfectly good calendar and whatever the rest of the codebase currently believed in.

Five rewrites, zero new capability. That's not engineering; that's rent.

Let me be honest about that history rather than merely aggrieved by it, because the honesty is where the argument lives. Nobody forced the fifth rewrite. No framework broke. The ecosystem's gravity simply made class components feel like technical debt, and most of us obeyed. The rewrites weren't even cheap, though each one was sold as "just the wrapper." The wrapper is where focus management,

keyboard handling, and popup positioning actually live, and that's how a "thin" rewrite reintroduces solved bugs. For the first several rewrites there was no alternative on offer, either: the platform had no component model to build on.

It does now. That's this book's pitch in one sentence: **the sixth rewrite is optional.** Make the component itself a thing the *browser* understands: its identity, markup, style, and behavior. Then the part that never changes finally gets to never change.

That's the argument in miniature. The rest of this chapter builds it up properly: where the component-as-unit-of-UI idea came from, why frameworks churn even when nobody involved is doing anything wrong, and what the platform actually offers instead. Because I don't want you taking this on faith, I'll also cover where the platform's model is a genuinely worse fit than a framework library. Let's start with how we got here.

1.1 The component era: from jQuery plugins to framework components

To be fair to 2009, the jQuery plugin was a component in spirit. It bundled behavior behind one call, it was reusable across projects, and an ecosystem of thousands of them existed because the model genuinely worked: drop in a script, call a function on a selector.

What it lacked was everything structural. A plugin had no lifecycle: it initialized once, and if the DOM it enhanced was later removed or replaced, its event handlers and timers leaked into the void. It had no ownership, either. Its CSS was global, its markup was whatever it found, and two plugins touching the same element negotiated by fistfight. It had no composition story: plugins didn't nest, didn't communicate, and didn't form trees. Every page was a flat list of enhancements applied to server-rendered HTML, plus duct tape.

The framework era's real discovery came through Backbone's views, Angular's directives, and most clearly React. It was the **component as the unit of UI**. A component is a named, self-contained thing. It owns its markup and behavior, declares its inputs, announces its changes, participates in a lifecycle, and composes into trees. An application stopped being pages-plus-enhancements and became a hierarchy of components. Nearly everything good about modern front-end practice falls out of that one modeling decision: reusability, testability, design systems as products, reasoning locally about one piece of UI.

That idea has now outlived every framework that helped discover it, which tells you something important: **the component model is the durable insight. Each framework's expression of it is a passing implementation.**

1.2 The framework churn problem and the case for the platform

I want to be precise here, because "frameworks churn" is usually delivered as a sneer and I don't mean it as one. Frameworks churn *because they are research programs* — they exist to explore the questions the platform hasn't answered yet. Exploration means revision. Revision means your code ages.

The record, with dates: Angular 2 shipped in September 2016 as a ground-up rewrite, incompatible with the AngularJS applications a generation of enterprises had bet on. AngularJS itself left long-term support on the last day of 2021. At that point, code written in 2015 needed a commercial support contract to stay patched. Vue 2 reached the same fate on the last day of 2023, with the same paid-extended-support coda. React never broke the world that way, but it restyled it. Hooks arrived in February 2019, and although class components still run today, the ecosystem, the documentation, the libraries, and the hiring pool all moved. Running unsupported idioms is its own slow tax.

Two hard breaks and an idiom shift, across the three biggest frameworks of the era, are not a law of nature. React's core API has genuinely never broken its users, and maybe your framework never will either. But that's precisely the problem with the bet: *at the moment you place it, you cannot know which trajectory yours will take.* None of this makes frameworks bad. It makes them **the wrong layer for your longest-lived UI investment.** A component library should not have its lifespan capped by the revision cycle of whichever framework was ascendant the year it started.

Now weigh the platform against that record. But let me state the platform's guarantee precisely, because it has fine print, and the fine print is load-bearing. The browser's backwards-compatibility discipline is the strongest in consumer software: HTML written in 1996 still renders, and `document.getElementById` has not broken an application in this century. The precise form of the promise is this: **once a standards-track API ships interoperably in every engine, it does not get broken.** Single-engine features enjoy no such protection, and the web-components story itself contains the proof. The original v0 custom elements API and HTML Imports shipped in Chrome alone. Real production applications were built on them, and then they got scrapped in redesign. The replacement v1 APIs were everywhere by 2018, and Chrome finally removed the old ones in early 2020. Early adopters paid real migration pain for building on a one-browser feature.

I'm not offering that episode as an asterisk to be excused. It's the rule to be learned, and this book applies it throughout: *interoperable is permanent; single-engine is provisional.* The v1 APIs cleared the interoperability bar years ago. That places them under the strongest compatibility umbrella the industry has. So the argument of this book, stated once and plainly:

Put the component layer on the platform, and let frameworks live above it.

Let me be precise about what that buys, because it is *not* immortality-without-maintenance. A

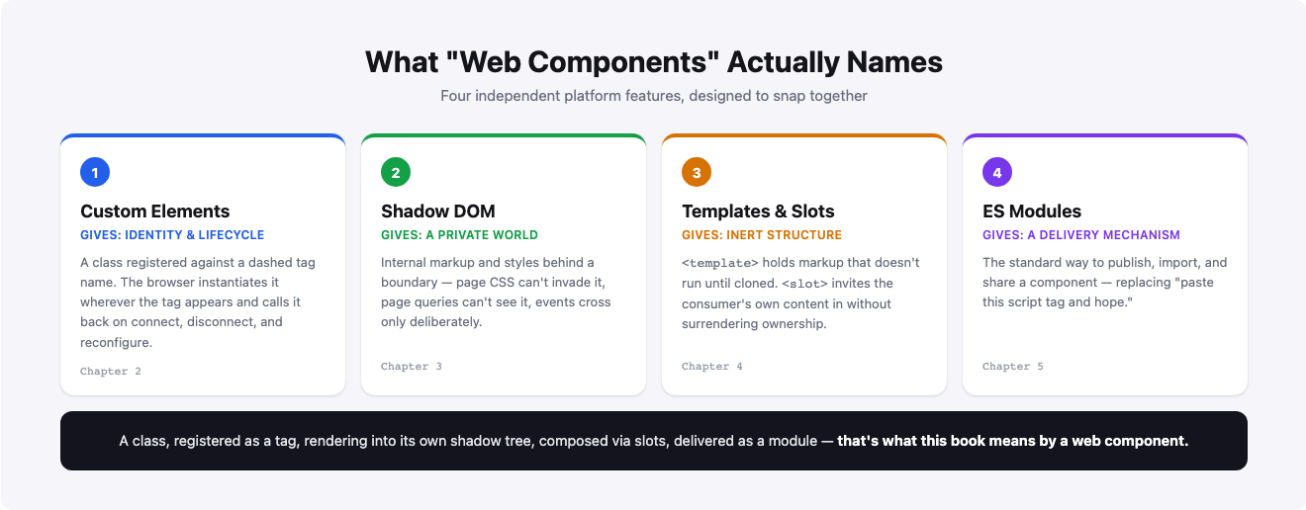
component library still has dependencies with revision cycles of their own: a build tool, a test runner, and, if you adopt one, a rendering library. What changes is *where churn lands*. A component's public interface is standard DOM: a tag, attributes, properties, events. So when a host application changes frameworks, the components don't get rewritten: the tags mean the same thing in the new world. And when a component's internals need updating, the change stays contained behind its boundary instead of smearing across every consumer. Churn stops being a multiplication across your codebases and becomes an addition inside one of them.

This is not an anti-framework position. I'd reach for a framework tomorrow to build a complex application. The claim is narrower and, I think, harder to argue with: the *components* belong to the layer whose revisions never break you — they are the layer you want to survive the next revision cycle.

1.3 What web components are: the four pillars

"Web components" names no single API. It's the collective label for four platform features that compose into a component model. Each gets its own deep treatment in the chapters ahead; here is the shape of the whole.

Figure 1.1 — the four pillars, and what each one gives a component:



Custom Elements (Chapter 2) give a component *identity and lifecycle*. You define a class and register it against a tag name containing a dash, like `<my-card>`. The browser then instantiates it wherever the tag appears, telling it forever after when it's added to a document, removed, or reconfigured. This is the pillar that makes a component a real HTML citizen rather than a div with aspirations. It works in `innerHTML`, in template engines, in Markdown renderers, in anything that can produce a tag.

Shadow DOM (Chapter 3) gives a component *a private world*. Its internal markup and styles live behind a boundary that page CSS can't invade, page queries can't see, and events cross only

deliberately. Encapsulation is what makes a component safe to drop into a page you don't control. It's what makes *distribution* possible.

Templates and slots (Chapter 4) give components *inert structure and composition*. `<template>` holds markup that doesn't run until cloned; `<slot>` lets a component invite the consumer's own content into designated openings without surrendering ownership of either side.

ES Modules (Chapter 5) give components *a delivery mechanism* — the standard way to publish, import, and share them. This pillar is easy to underrate until you remember what it replaced: components used to have no distribution story beyond "paste this script tag and hope."

The four are independent; you can use any without the others. But they're designed to snap together, and the composed whole is what this book means by a web component: *a class, registered as a tag, rendering into its own shadow tree, composed via slots, delivered as a module*.

1.4 Where web components shine

Three problem shapes fit web components so well that they've become the default answer, plus two more that deserve the label.

Design systems. UI decisions that must reach every team building product, including the teams and codebases you don't control. The big one — and the strongest version of the whole argument, so it gets its own section, next.

Micro-frontends. When separate teams own separate slices of one page, custom elements are the natural seam. Each team ships elements, the shell composes tags, and shadow DOM keeps eleven teams' CSS out of each other's pockets. The contract is the DOM itself: properties down, events up. It's the thinnest inter-team API that can work.

Embeddable widgets. A chat bubble, a checkout button, a video embed — anything that must work on *pages you've never seen*. This is the hostile-environment case: unknown CSS, unknown frameworks, unknown other widgets. Encapsulation isn't a nicety here; it's survival gear. A single custom tag is the friendliest install instruction ever written.

Long-lived application shells. An application expected to evolve for a decade-plus, where betting the shell on the platform means never facing a forced migration.

Incremental modernization. The unglamorous one, and on the ground the one I see most often. A custom element doesn't need a build system, a root component, or a router; it's just a tag. That means you can drop one modern, encapsulated component into a fifteen-year-old server-rendered page, then another, then ten more, without ever scheduling The Rewrite. The platform meets legacy code where it lives.

One scoping note, so this section sells only what the book delivers. Micro-frontends and embeddable widgets share nearly all of their mechanics with the design-system case: encapsulation, event contracts, packaging, registry hygiene. So the techniques transfer. But maybe you want a book about micro-frontend *architecture*, covering shell orchestration, independent deployment, and cross-team versioning. That's a different book, and I'd rather tell you now than disappoint you later.

1.5 Why a design system belongs on web components

Everything so far argues that web components are a good way to build reusable UI. This section argues something narrower and more consequential, because it's the thesis the back half of this book rests on: **for a design system specifically, I think the platform is the right default, and the reasons are different from the general case.**

Start with what a design system *is*, since the term covers three different things. A design system is not a component library, though it contains one. It's an organization's UI decisions about spacing, color, type, interaction, and accessibility standards: made once, encoded as artifacts, and distributed to every team that builds product.

That definition establishes distribution to many *teams*. It does not, by itself, establish many *frameworks*: a company with forty React apps distributes to forty teams and satisfies it completely. So the multi-framework premise has to be earned empirically rather than smuggled in by definition. Here is the empirical claim: over a design system's lifespan, consumer heterogeneity is the normal end state. An acquisition arrives on a different stack. The marketing site stays server-rendered for reasons nobody will overrule. A profitable checkout flow is too risky to migrate. And the system is expected to outlive the standardization decree itself, because decrees have framework-version lifespans and brands don't. N rarely stays 1; the question is whether you planned for that or got surprised by it.

Granting that, three properties push toward the platform — and, to be honest about it, one pushes hard the other way.

It must outlive its consumers' frameworks

A design system is a decade-scale asset. Brands, spacing scales, and accessibility standards outlive framework generations. Build the system in React and you have quietly bet that React's revision cycle stays comfortable for as long as your brand exists. Build it on the platform and the bet gets much smaller.

One caveat, since this is where the platform's story is most often oversold: a tag name has no version, and that cuts both ways. `customElements.define()` is a global registry keyed by tag name, and it throws on a duplicate. So two versions of `<prism-button>` cannot coexist on one page the way two versions of a React button coexist in a bundler trivially. That's a real constraint on how you ship

breaking changes and a real hazard when two consumers disagree about versions. Scoped registries are the fix — Chapter 11 is devoted to them. Longevity is a genuine advantage; frictionless versioning is not.

It must be consumed by codebases you don't control

A design system team ships to consumers it doesn't own: the React app, the Angular admin console, the server-rendered marketing site, the acquisition still on jQuery, the Salesforce customization somebody built in 2019. A framework-native library serves one of those well and the rest badly. That's not "abandons": a React library can be wrapped as a custom element for the stragglers. But the wrapping is work somebody pays for.

Weigh the honest alternatives. *Build it N times* multiplies implementation, testing, accessibility audit, and documentation by N. It also guarantees drift. The React button and the Angular button are maintained by different people on different sprints, and by month eight they disagree about focus rings. *Ship tokens only, let teams build their own components* is the most common real compromise, and its cost distribution explains its popularity: cheap for the system team, expensive for the organization. Values get standardized; behavior doesn't. You end up with consistent color and inconsistent interaction, and no single place to fix an accessibility bug.

The alternative that deserves real respect is a **headless behavior library plus tokens** — a library that ships only behavior and accessibility wiring, with no visual output of its own, so each framework can style it however it likes. React Aria, Radix, and Ark are the well-known examples. Behavior, keyboard maps, focus management, and ARIA wiring come from a shared, deeply specialized library. Tokens carry the visual decisions, and each product composes its own presentation. On compound-widget accessibility this approach is genuinely excellent, better than most hand-rolled web components will be. Its limitation is precisely the axis this section is about: these libraries are framework-scoped. React Aria is React. Adopt it and your N>1 problem is untouched, which is why Adobe maintains React Aria *and* Spectrum Web Components. Two answers to two different questions.

Its components must survive contact with hostile CSS

A design system's components land in pages the system team has never seen, carrying a decade of stylesheets, a utility framework, a vendor widget, and at least one `!important` nobody dares delete.

Shadow DOM makes that survivable, precisely because page CSS cannot *select* your internals. Inheritance still flows in; a `line-height` on `body` reaches your button's text. And the host element remains fully styleable from outside. That's a boundary, not a bunker. What it buys a design system is transferability: a component's rendering in isolation predicts its rendering inside a consumer page. That's what makes a visual snapshot mean something beyond the page it was taken on. Framework libraries run visual regression testing — automated screenshots compared against a saved baseline, so an unintended visual change gets flagged — perfectly well; tools like Storybook and Chromatic long

predate any of this. What they can't promise is that the result survives contact with someone else's stylesheet.

The same boundary forces theming to be designed rather than assumed: deliberate, documented seams — custom properties for values, parts for surfaces, custom states for conditions. That's a cost, and also a feature. The alternative, where any consumer restyles anything, is how design systems drift into decoration.

Its API must be documentable and machine-readable

A design system is only as good as its contract, and web components express theirs in the DOM's own vocabulary — attributes, properties, events, slots, custom properties, parts.

Be careful about what that does and doesn't win. A TypeScript React library has a thoroughly machine-readable contract too, and its types can express far richer shapes than a string attribute ever can: a value restricted to a fixed set of options, a type assembled from other types, the exact shape a callback function is expected to have. The platform's advantage isn't that a description exists; it's that the description is *portable*. One custom-elements manifest — a JSON file, generated from your source, that describes every component's attributes, properties, events, and slots (more on it in Chapter 17) — feeds editor autocomplete, the documentation site, and the coding agents now writing much of product UI. The manifest is itself a community convention, still pre-1.0, not a platform guarantee. A framework library describes itself in one framework's idioms. A platform library describes itself in terms every tool already parses.

The property that points the other way

Now the one I'd be dishonest to leave out, because it bites hardest exactly here. A design system is disproportionately form controls and compound widgets: button, input, select, combobox, dialog, tabs, menu. Those are where the shadow boundary costs you. ARIA relationships can't cross it, so a consumer's `<label for>`, `aria-describedby`, or `aria-controls` cannot reach your internals. Chapter 10 covers the workarounds, and `ElementInternals` recovers much of the form story, but it remains the platform's roughest edge and the headless libraries' clearest win. If your system is mostly complex a11y-critical widgets and mostly one framework, that consideration deserves to outrank the other three.

The honest counter-case

Here is the strongest opposing position, stated the way its advocates state it: *"We have forty React apps and three non-React surfaces. We build on React Aria, so we get best-in-class compound-widget accessibility, native consumer DX, mature SSR, and a hiring pool. We ship a custom-element bridge for the stragglers. And we win on adoption — the only metric that decides whether a design system succeeds."*

That's a good argument, and the adoption point is the sharpest part of it: a system product teams don't reach for is worthless regardless of its architecture. The platform's honest reply is that the bridge is doing more work than it sounds. Behavior authored against React's rendering model rarely bridges cleanly, and the forty/three ratio is a snapshot, not a trend. But if the ratio holds and the widgets are gnarly, I'd take the React answer without embarrassment.

Two more scoping notes. If your organization truly has one application and one framework, "distributed" collapses and so does this entire section. You want a component folder with good conventions, not a design system. And if you're mostly one framework with a long tail, the answer is usually a platform core with thin framework wrappers on top — Adobe, Microsoft, and SAP all ship exactly that. Choosing the platform for the core doesn't oblige you to make React developers write `addEventListener`.

What the rest of this book tries to show is that these properties compound. Encapsulation makes visual snapshots transferable. A DOM-level contract makes the manifest possible, and the manifest feeds the docs, the tooling, and the agents. Platform longevity makes all of it worth building once. Part IV builds Prism DS to make that compounding possible, and Parts V and VI cash it in — which is why the sample system is architected the way it is rather than the shortest way.

1.6 Where they don't: honest trade-offs

A book that only argues *for* its subject is marketing. Here is the case against, as fairly as I can make it — with pointers to where this book confronts each item rather than waving at it.

Web components are not a framework, and the gap is real. You get identity, encapsulation, lifecycle, composition, and nothing else. No state management, no router, no data layer, no opinion about how an *application* hangs together. For a component library that's mostly fine, but a team expecting React-the-ecosystem and receiving four DOM primitives is going to have a bad quarter. The authoring experience deserves its own warning: vanilla web-component code is imperative DOM programming. If you've spent years in JSX, Part II will occasionally feel like a time machine. That complaint is legitimate, and it's the strongest argument for putting a small rendering library on top. Lit is the best-known one: a small library that lets a custom element's render method be written as a declarative, JSX-like template instead of hand-built DOM calls, while the element underneath stays a real, standard custom element. Lit and its neighbours exist to restore declarative templates without abandoning the platform. This book stays on the platform so that the design-system arguments never depend on one library's release schedule. Scope your expectations either way: this is a component technology, not an application architecture.

Encapsulation cuts both ways. The boundary that protects your component from the page also stops the page's design language at the door. Utility-CSS workflows, org-wide theme sweeps, the quick

one-off override in a product sprint — all of it halts at the shadow root. Consumers can restyle only what the author deliberately exposed through custom properties and parts. This is the flip side of the book's central selling point, and it's the friction that app teams who *want* global styling control hit first and loudest — Tailwind-era teams especially. A component author designs the styling API on day one or hears about it forever.

Your test tooling has opinions. A DOM full of shadow roots defeats naive `querySelector` assertions and the jsdom-era habits most test suites are built on. The tooling has caught up unevenly, and cross-boundary queries remain awkward in much of it. Chapter 20 chooses a stack that works with the grain instead of against it, but if your organization's testing conventions are deeply framework-native, budget for the mismatch.

Instances aren't free. Every component instance carries its own shadow tree. At ten components on a page this rounds to nothing; at ten thousand table cells it's measurable memory and startup work. `adoptedStyleSheets` eliminates the *style* duplication, but the per-instance tree is yours either way. A many-instance grid deserves deliberate design rather than assuming one rendering strategy fits every table.

The DOM's interface is stringly typed at the edges. Attributes are strings; rich data wants properties. Keeping the two coherent is a discipline the platform makes *possible* but not *automatic*. Chapter 6 exists because this genuinely trips people, and the boolean-attribute trap there catches almost everyone once.

Server-side rendering arrived late. Framework SSR is a decade old; the platform's answer, Declarative Shadow DOM, only became usable in every engine in early 2024. It's a good answer, but the surrounding tooling is years younger than Next.js, and it shows. The adjacent SEO worry mostly dissolves with DSD, since the content is right there in the HTML. But "mostly" is doing quiet work in that sentence.

Accessibility has one genuine gap. ARIA relationships can't point across shadow boundaries — a label out in the page can't reference an input inside a shadow tree by ID. I won't pretend it's solved; it's the roughest edge the platform still carries.

A component can exist before its definition arrives. The upgrade gap is real, structural, and either styled around or engineered away. Frameworks don't have this failure mode; you should know you're buying it.

Ecosystem gravity is real. React's ecosystem is larger by an order of magnitude: component marketplaces, hiring pools, Stack Overflow mass. Choosing the platform means building more things yourself. One old objection has aged out, though. "React doesn't play well with custom elements" was true for years and shaped a generation of opinion. But React 19, shipped December 2024, made custom elements first-class, and every major framework now scores a perfect 100 on the custom-elements-

everywhere interop suite. That objection is simply out of date now.

And sometimes the honest answer is: don't. One product, one team, one framework, no reuse ambitions beyond the app? Framework-native components are simpler, better-tooled for that world, and entirely correct. Web components earn their keep where UI investment must *outlive or out-span* a single stack — that's the design-system case, the widget case, the decade-scale case. If none of those describe you, close this book with my blessing. (You won't be able to tell anyone what happens in Chapter 3, though.)

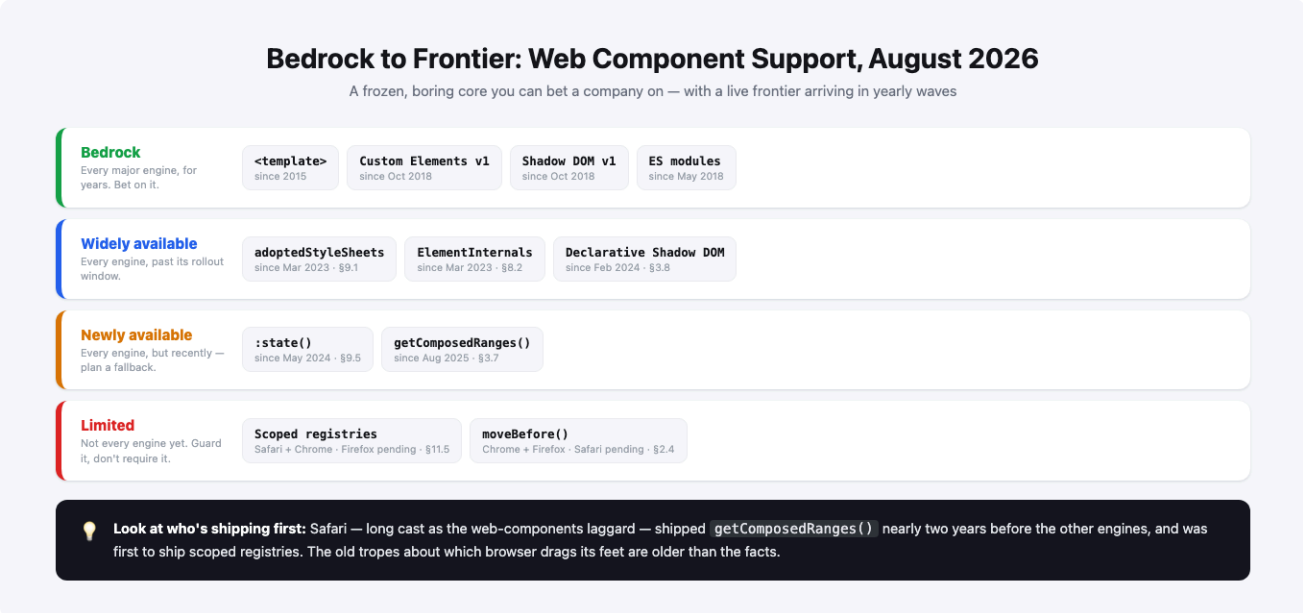
1.7 Browser support and the post-polyfill landscape

For years, every web-components talk carried a slide of caveats: polyfills for this browser, partial support in that one. That era is over. It's worth being precise about when it ended and what "over" means, because a lot of the internet's received wisdom on this topic was written while it was still true.

Three dates draw the map. The **v1 core** has worked in every major engine since Firefox completed the set in **October 2018**: custom elements, shadow DOM, templates, modules. The **polyfill era ended in June 2022**, when Internet Explorer 11 was finally retired. The venerable `@webcomponents/polyfills` project lives on in maintenance mode, but if your support matrix is evergreen browsers, you will never load it. And the **frontier is still moving**, which is the healthiest thing on this list, because it means the platform is being invested in, not merely maintained.

Here's the honest status as I write, in August 2026 — grouped by tier rather than a flat list, because the tier is the thing worth remembering, not any single date:

Figure 1.2 — bedrock to frontier:



Two observations worth carrying out of that figure. First, the shape of a healthy platform: a frozen, boring core you can bet a company on, with a live frontier arriving in yearly waves. Everything Parts I and II *require* sits in the top half. Where a chapter reaches lower, the feature is an enhancement, and flagged as one in place.

Second, and this one upends a decade of conventional wisdom: **look at who's been shipping first.** Safari, long cast as the web-components laggard, shipped `getComposedRanges()` nearly two years before the other engines and was *first* to ship scoped custom element registries, in September 2025. The tropes about which browser drags its feet are older than the facts. One limb really is dead, though, and it stays dead: customized built-in elements, `is="…"`, which Safari rejected permanently.

1.8 Who uses them: come and see

Arguments from authority are weak; arguments from *verifiable production behavior* are stronger. Everything in this section can be checked from your own DevTools, and the hands-on at the end of this chapter hands you the exact snippet these numbers were gathered with, so you can rerun the survey yourself. But let me be straight about what this evidence can and cannot prove. Giant-scale adoption settles exactly one question: whether these APIs bear production load. It does not settle whether they fit *your* team's constraints. This section is the existence proof, nothing more; it happens to be a strong one.

GitHub ships hand-rolled custom elements across github.com. `<relative-time>` renders every timestamp you've ever read there, alongside `<clipboard-copy>`, `<include-fragment>`, `<details-dialog>`, and a dozen siblings, published openly in the `github-elements` repo. But the more instructive fact is what GitHub does with its *newer*, React-built surfaces: it mounts them inside custom elements. When I inspected a repository page while writing this chapter, the React code view was sitting inside `<react-app>` and `<react-partial>` elements — the framework living *above* the platform layer. Nobody at GitHub had to choose between web components and React; the architecture holds both.

YouTube has been a custom-elements application since its 2017 redesign, and still is: the entire desktop app boots as `<ytd-app>`. On the day I checked, its homepage contained sixty-five distinct defined custom elements. Its Polymer heritage makes it the most instructive case here, and not in a purely flattering way, so let me score it honestly. Polymer the *library* went into maintenance years ago, and YouTube still carries that runtime today: a real, ongoing cost, the same kind of unsupported-dependency tax I charged against AngularJS earlier. What YouTube *never* faced is the other, worse kind of churn: the Angular-2-style cliff, where staying supported means rewriting the application. Because its components' public interfaces are standard DOM, the escape route is incremental. You migrate internals element by element while the site ships every day, no big bang required. Contained churn instead of multiplied churn, running at planetary scale, cost sheet included.

Adobe builds Spectrum Web Components, the coded expression of its design system — and when Adobe put *Photoshop itself* on the web, the UI was built from Lit-based web components. **Salesforce** went further than anyone: Lightning Web Components isn't a component library but the actual application framework of the Lightning Platform, which means a meaningful slice of the world's CRM screens are custom elements. **SAP** ships UI5 Web Components as the standards-based face of its enterprise UI stack. All three are actively developed as I write — these are load-bearing corporate commitments, not hackathon leftovers.

Microsoft is the story most often told wrong, so precisely: the FAST project narrowed to its core engine (`fast-element`, which shipped a new major version in June 2026), and Fluent UI Web Components reached a stable 3.0 the same month. Most tellingly, the Edge team has been methodically *replacing React with web components inside the browser's own UI*, reporting the Browser Essentials panel got 42% faster for it. When a browser vendor rebuilds its own chrome on these APIs, "is this production-ready?" stops being an interesting question.

The browsers themselves closed the loop: Chrome's DevTools mandates web components with lit-html rendering for its UI, and Firefox builds its `about:` pages on a Lit extension called `MozLitElement`. The platform's stewards build *on* the platform. Beyond the giants, the pattern repeats through banks (ING's Lion), automakers (Porsche's design system), and component vendors (Vaadin; the Font Awesome team's Web Awesome). In the aggregate, Chrome's own telemetry reports that code on **roughly one in five page loads** registers at least one custom element.

You are, statistically speaking, already a web-components *consumer*, several times a day. Whether you should become an *author* is a question about your problems' shapes — and the rest of this book is my answer for the shapes that fit.

1.9 Hands-on: an expedition

No component to build yet — Chapter 2 starts that. Instead, verify this chapter's boldest claims yourself, because a book that opens by telling you not to trust framework fashion shouldn't ask you to trust its author either.

1. **Open a major site you use daily** in a regular browser tab. Both github.com and youtube.com work well.
2. **Open DevTools and switch to the Console panel.** In Chrome or Edge, right-click anywhere on the page and choose "Inspect," then click the "Console" tab at the top of the DevTools pane if it isn't already selected.
3. **Paste this snippet into the console and press Enter.**

```
[...new Set(
  [...document.querySelectorAll('*')]
    .map((el) => el.tagName.toLowerCase())
    .filter((tag) => tag.includes('-')),
)].filter((tag) => customElements.get(tag));
```

You should see an array of lowercase, hyphenated tag names printed below the prompt: every distinct hyphenated tag on the page, filtered down to the ones with a registered definition. Running it while writing this chapter, on August 16, 2026, turned up a GitHub repository page carrying **eleven** defined custom elements, including `relative-time`, `clipboard-copy`, and the React-hosting `react-app`. youtube.com carried **sixty-five**, from `<ytd-app>` on down.

Here is the actual console output, from re-running that exact snippet on `github.com/facebook/react` on August 27, 2026:

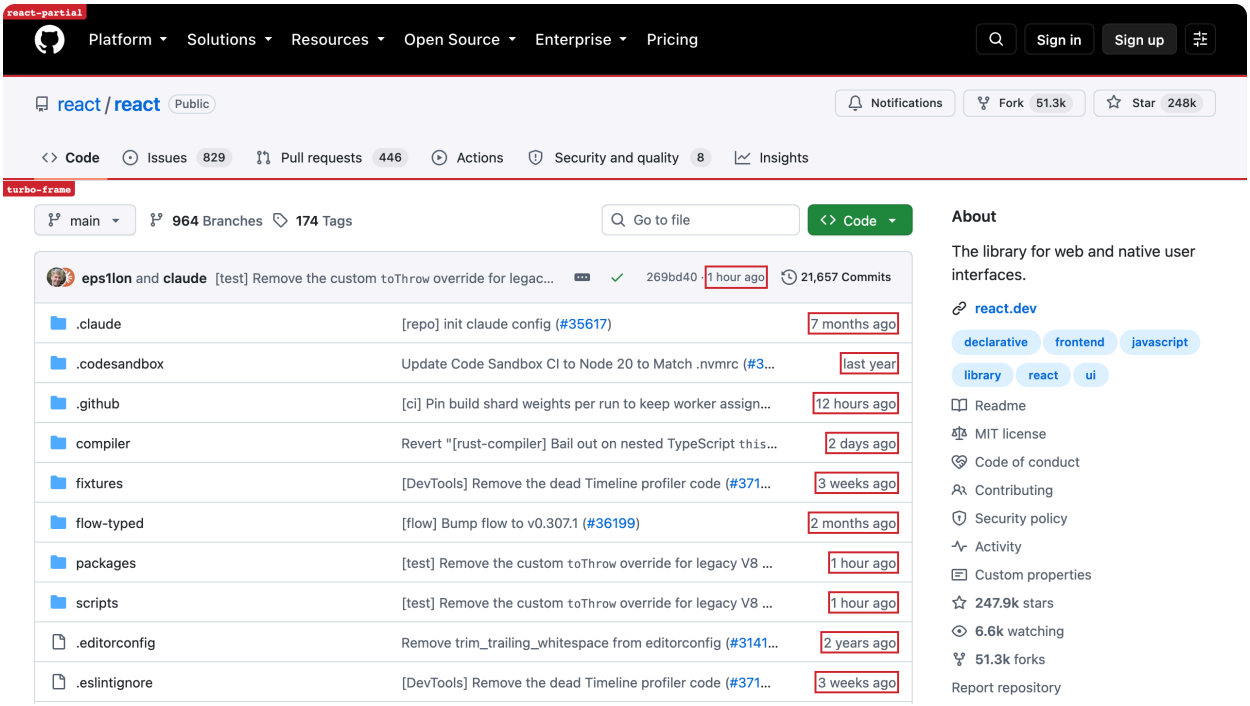
```
['action-list', 'action-menu', 'anchored-position', 'clipboard-copy',
 'cookie-consent-link', 'focus-group', 'ghcc-consent', 'include-fragment',
 'react-app', 'react-partial', 'relative-time', 'tool-tip', 'turbo-frame']
```

Thirteen, not eleven — eleven days later, on the same site. That drift is the point rather than a correction: nobody shipped a press release about it, and the page looks identical. youtube.com moved too, from sixty-five to **sixty-six**. Your own counts will differ again, and on either site the array should be non-empty.

4. **Paste this follow-up snippet into the same console** to outline every one of those elements in place, right on the live page:

```
[...document.querySelectorAll('*')]
  .filter((el) => customElements.get(el.tagName.toLowerCase()))
  .forEach((el) => {
    el.style.outline = '2px solid red';
    el.style.outlineOffset = '1px';
  });
```

Figure 1.3 — the same GitHub page, produced by exactly that snippet. The boxed timestamps down the commit column are each a `<relative-time>`:



That figure is the argument of this chapter in one screenshot. Nothing on that page announces itself as a web component. The timestamps are the tell: each one is a custom element that upgrades itself, formats a machine-readable date into human prose, and keeps updating — sitting inside a page most people would describe, without hesitation, as "a React app."

5. **Switch to the Elements panel** and use the element picker (the cursor icon top-left of the DevTools pane, or `Cmd+Shift+C` / `Ctrl+Shift+C`) to click one of the tags step 3's list printed.

6. **Look inside that element's node in the Elements tree for a child labeled `#shadow-root`.** If you see one, click the triangle to expand it — you're looking at shadow DOM in the wild. Note the text in parentheses next to it: it reads either `(open)` or `(closed)`.

   Not every custom element has one, and which ones do is worth noticing. On that same GitHub page, exactly three of the thirteen carried an open shadow root: `tool-tip`, `include-fragment`, and `relative-time`. The other ten had none at all. On youtube.com the split is starker: `<ytd-app>` is defined, present, and renders entirely into the light DOM with no shadow root anywhere in the sampled tree. That's the independence of the four pillars showing up in production rather than in a diagram: custom elements and shadow DOM are separate pillars, and a great many real components use the first without the second.

7. **On github.com specifically, find a `<react-app>` element in the Elements panel** (use the element picker on any repository page, or `Cmd+F` / `Ctrl+F` inside the Elements panel to search for

`react-app`) **and expand it.** You should see ordinary React-rendered markup nested inside — no shadow root this time, just a plain subtree. I confirmed both halves of that on the run above: `react-app` was present, its `shadowRoot` was `null`, and its children were plain light-DOM nodes. That's the layering this chapter describes: a framework mounted *above* the platform's custom-element boundary, not replacing it.

8. **Repeat step 3's snippet on four more sites you visit regularly**, keeping a tally of how many defined custom elements each one reports. In my own experience the count is rarely zero, and on the rare page where it *is* zero, the page usually either predates the component era or is all-in on a single framework — and even then, a stray `<something->` wrapper is increasingly likely to turn up if you keep looking.

What to remember

- **The component model is the durable insight of the framework era**; each framework's expression of it has a revision cycle. Web components make the model a platform primitive under the precise guarantee that matters: interoperable is permanent, single-engine is provisional — and churn lands inside the component instead of across every consumer.
- **This is not anti-framework**: frameworks live *above* the component layer. GitHub mounts React inside custom elements; that layering is the book's thesis in one DOM tree.
- **Four pillars**: custom elements (identity, lifecycle), shadow DOM (encapsulation), templates and slots (structure, composition), ES modules (delivery).
- **The sweet spots**: design systems, micro-frontends, embeddable widgets, decade-scale shells, incremental modernization. The honest pass: single-team, single-framework products with no reuse ambitions.
- **A design system is the strongest case** — it must outlive its consumers' frameworks, reach codebases you don't control, survive hostile CSS, and expose a portable contract — with one property pulling the other way: cross-root ARIA, which bites hardest on the form controls a system is mostly made of.
- **The core has been stable everywhere since 2018**, the polyfill era ended with IE11 in 2022, and the frontier (DSD, custom states, scoped registries) is still arriving — check current status before leaning on anything shipped after 2023, and don't take my word for any of it: one console snippet shows you the custom elements on any page you visit.

Chapter 2: Custom Elements 101

This chapter covers

- Registering an element with the browser
- Autonomous elements vs. customized built-ins
- Naming rules, and why the hyphen matters
- The lifecycle callbacks, in order
- The upgrade gap: elements that exist before they work
- Building `<my-card>`

Introduction

Custom elements are the pillar most people think they already understand, because on the surface it's just a class and a `define()` call. What trips people up isn't the API surface. It's the ordering. When exactly does your constructor run relative to the attributes already sitting on the tag? What happens to an element the parser meets before your script has even loaded? Those questions don't show up in a quick demo. They show up in production, on a slow connection, for a user nobody on the team happens to be.

Here is a bug that ships somewhere every week. A team puts its first custom element into production. It works perfectly on every development machine, and renders nothing at all for the slice of users on slow connections.

The component is fine. The HTML is fine. What differs is timing. Those users' browsers parse `<user-badge>` long before the script defining it arrives, and until it lands, the page shows an empty gap where a name should be. On a developer's laptop, with a warm cache, the script wins that race every time. On a train in a tunnel, it doesn't. Nobody on the team sees the bug, because nobody on the team is its victim.

That bug is the right way into this chapter, because it teaches the thing custom elements are really about. A custom element isn't a class you write and then use. It's a class you *register*. Everything interesting follows from how registration works: when it renders, what it can do in its constructor, whether it exists at all when the parser reaches it. Get the mental model right and the whole API feels obvious. Get it wrong and you'll spend an afternoon wondering why your constructor can't see its own attributes.

So let's start with the registry, since everything else in this chapter follows from how it works.

2.1 Defining a custom element in the registry

Every browsing context has one global registry of custom elements, reachable as `window.customElements`. It maps a tag name to a class:

```
// component/card.component.js
class Card extends HTMLElement {
  connectedCallback() {
    this.textContent = 'Hello, World!';
  }
}

customElements.define('my-card', Card);
```

That's a complete, working custom element. Put `<my-card></my-card>` in a page, load this script, and the browser does the rest.

The mapping is global, permanent, and exclusive. You cannot define `my-card` twice — the second call throws a `NotSupportedError`. You cannot un-define it. You cannot swap the class out later. Once `my-card` means this constructor, it means this constructor for the lifetime of the document.

That permanence is worth sitting with, because it's the source of a real constraint you'll hit the moment two versions of a component library end up on the same page. I'll come back to it in Chapter 11. For now, just note that the registry is a one-shot, first-writer-wins resource.

Beyond `define()`, the registry has three methods worth knowing, and most developers I've taught have used none of them.

`customElements.get(name)` returns the constructor, or `undefined` if the name is unclaimed. Use it to check before defining, which matters if your component might be loaded twice:

```
if (!customElements.get('my-card')) {
  customElements.define('my-card', Card);
}
```

`customElements.whenDefined(name)` returns a promise that resolves once the name is registered. This is the honest fix for a class of race conditions where you need to touch a component's API from outside:

```
await customElements.whenDefined('my-card');
document.querySelector('my-card').refresh();
```

Without that await, `refresh()` might not exist yet. The element in the DOM is still an

`HTMLUnknownElement`-ish placeholder with none of your methods on it.

`customElements.upgrade(root)` forces the browser to upgrade any already-defined custom elements inside a subtree, synchronously. You rarely need it, but you need it badly when you do. Elements created in a document that isn't being rendered don't get upgraded on their own, whether that document came from `DOMParser` or from `document.implementation.createHTMLDocument()`. Build markup in a detached document and its custom elements sit there inert until you call `upgrade()` or move them into the live document.

2.2 Autonomous custom elements vs. customized built-ins

The spec describes two kinds of custom element, and one of them is a trap.

An **autonomous custom element** extends `HTMLElement` and introduces a brand-new tag. Everything so far has been one of these:

```
class Card extends HTMLElement { /* ... */ }
customElements.define('my-card', Card);
```

```
<my-card></my-card>
```

A **customized built-in** extends a specific existing element and augments it, using the `is` attribute:

```
class FancyButton extends HTMLButtonElement { /* ... */ }
customElements.define('fancy-button', FancyButton, { extends: 'button' });
```

```
<button is="fancy-button">Click me</button>
```

Customized built-ins look strictly better on paper. You inherit everything `<button>` already does instead of rebuilding it: form submission, keyboard activation, the correct implicit ARIA role, focus behavior. I wanted them to win.

They didn't. WebKit declined to implement them, and that position has held for years. Any customized built-in you write simply won't upgrade in Safari. You get a plain `<button>` with an ignored attribute. Polyfills exist and are unpleasant.

So my advice is blunt: **use autonomous custom elements.** Everything in this book does. The cost is that you'll rebuild button semantics by hand later. I won't pretend that's free — it's most of a chapter. But a component that silently degrades in one major browser isn't a component you can put in a design

system.

2.3 Naming rules and conventions

The dash isn't a style choice. It's the entire forward-compatibility strategy for HTML.

A valid custom element name must contain at least one hyphen, must start with a lowercase ASCII letter, and must not contain uppercase letters. The hyphen is the guarantee. HTML promises never to ship a built-in element with a hyphen in its name, so a hyphenated tag is permanently yours. `<my-card>` can never collide with a future standard element the way `<card>` might.

A short list of names is reserved despite containing hyphens, because they already exist in SVG and MathML: `annotation-xml`, `color-profile`, `font-face`, `missing-glyph`, and a few relatives. You will almost certainly never pick one by accident.

Break the rules and `define()` throws a `SyntaxError` immediately, which is the kindest failure mode in this whole API. Compare it to what happens if you *don't* use a custom element name at all: nothing throws. The browser treats `<card>` as an unknown element, styles it as inline, and never upgrades it. Silence.

Two conventions worth adopting, neither required:

Prefix by system, not by type. `<prism-button>`, not `<button-prism>`. Prefixes cluster alphabetically in autocomplete and make it obvious at a glance which components came from where. That gets genuinely useful once a page carries components from three different libraries.

Name the class after the tag. `PrismButton` for `<prism-button>`. Obvious, and universally ignored the moment someone writes a `Card` class for `<my-card>` — which is exactly what this chapter's own examples do below, kept that way on purpose so the class name reads naturally in isolated snippets. In the real system this book builds later, tag and class names match.

2.4 Lifecycle callbacks

A custom element has six lifecycle callbacks. Here they are, and then the details that actually matter:

CALLBACK	FIRES WHEN
`constructor`	The element is created. Not yet in the DOM.
`connectedCallback`	The element is inserted into a document.
`disconnectedCallback`	The element is removed from a document.
`attributeChangedCallback`	An *observed* attribute changes.
`adoptedCallback`	The element moves to a new document.
`connectedMoveCallback`	The element is moved via `moveBefore()`. Limited availability.

The constructor is more restricted than you expect

The rules are: call `super()` first, and then do almost nothing.

Specifically, you must not inspect attributes, must not inspect or add children, and must not add attributes to yourself. This isn't guidance. Violating it can throw, and will certainly break things in ways that are hard to trace.

The reason is `document.createElement('my-card')`. That call runs your constructor on an element with no attributes and no children, because none have been supplied yet. If your constructor assumed `this.getAttribute('message')` returned something, it now has to handle `null` on every creation path. The spec removes the ambiguity by forbidding the access outright.

What the constructor *is* for: setting up internal state, and attaching the shadow root.

```
class Card extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: 'open' });
    this._clicks = 0;
  }
}
```

`connectedCallback` runs more than once

This is the mistake I see most often. `connectedCallback` is not "on mount, once." It fires every single time the element is inserted into a document. Moving an element in the DOM is, mechanically, a removal followed by an insertion:

```
document.body.appendChild(existingCard);
// disconnectedCallback, then connectedCallback. Both fire.
```

So anything you do in `connectedCallback` must be safe to do repeatedly, or must be guarded. Add an event listener there without removing it in `disconnectedCallback` and every move through the DOM leaves another listener behind.

The pairing rule is simple and worth applying mechanically: **whatever `connectedCallback` starts, `disconnectedCallback` stops.** Listeners, intervals, observers, subscriptions, in-flight fetches.

```
connectedCallback() {
  this.render();
  this.addEventListener('click', this._onClick);
  this._timer = setInterval(() => this.tick(), 1000);
}

disconnectedCallback() {
  this.removeEventListener('click', this._onClick);
  clearInterval(this._timer);
}
```

Note that `this._onClick` has to be a stable reference for `removeEventListener` to find it. That means a bound method assigned once, not an inline arrow function created fresh in `connectedCallback`. That's an easy leak to write and a hard one to notice.

`connectedMoveCallback` and `moveBefore()`

The disconnect-reconnect-on-move behavior has been a genuine problem for years. Moving an `<iframe>` reloads it. Moving a `<video>` interrupts playback. Moving a component tears down and rebuilds state that didn't need to change.

`moveBefore()` fixes this. It relocates a node without disconnecting it, preserving state. Instead of the disconnect/connect pair, your element gets `connectedMoveCallback`:

```
document.body.moveBefore(cardElement, targetElement);
// connectedMoveCallback fires. Not disconnected/connected.
```

This is newer API with limited availability at the time of writing, so treat it as an enhancement rather than something to depend on. If you implement `connectedMoveCallback`, keep it to genuinely move-specific work — rebinding a document-scoped resource, say. Let the connect/disconnect pair remain correct on its own for browsers that don't support it.

2.5 Reacting to attribute changes

Attribute changes don't notify you by default. You opt in, per attribute, with a static `observedAttributes` list:

```
class Card extends HTMLElement {
  static observedAttributes = ['message'];

  attributeChangedCallback(name, oldValue, newValue) {
    this.render();
  }
}
```

Only `message` triggers the callback. Change `title` or `data-anything` and nothing fires. The list is read once, when the element is defined — you cannot add to it later.

Now the part that surprises people, and the reason I've given this its own section rather than folding it into the lifecycle table.

`attributeChangedCallback` fires before `connectedCallback`.

When the parser encounters `<my-card message="Hello">` and the element upgrades, the browser calls `attributeChangedCallback` once for each observed attribute already present, and *then* calls `connectedCallback`. That's the defined order. It exists so your element sees its initial configuration before it's asked to do anything.

It also means this is broken:

```
// ⚠️ Broken — do not copy
class Card extends HTMLElement {
  static observedAttributes = ['message'];

  connectedCallback() {
    this.attachShadow({ mode: 'open' });   // shadow root created here
  }

  attributeChangedCallback(name, oldValue, newValue) {
    this.shadowRoot.textContent = newValue; // ...but this runs first
  }
}
```

Loading that version for real produces exactly the failure the ordering rule predicts:

```
Uncaught TypeError: Cannot set properties of null (setting 'textContent')
```

`attributeChangedCallback` runs before `connectedCallback`, so `this.shadowRoot` is `null` and the whole thing throws on the first element the parser touches.

The fix is the one I've been quietly applying all chapter: **attach the shadow root in the constructor.** Then it exists before any callback can need it, and the ordering stops mattering.

```
constructor() {
  super();
  this.attachShadow({ mode: 'open' });
}
```

One more detail. `attributeChangedCallback` fires whenever the attribute is *set*, not when its value changes:

```
card.setAttribute('message', 'Hello');
card.setAttribute('message', 'Hello'); // fires again, oldValue === newValue
```

If your callback does real work like re-rendering or refetching, guard it:

```
attributeChangedCallback(name, oldValue, newValue) {
  if (oldValue === newValue) return;
  this.render();
}
```

2.6 Moving a component between documents

`adoptedCallback(oldDocument, newDocument)` fires when an element moves between documents. Most developers never see it fire, conclude it's academic, and move on. It isn't. The situations are just specific:

- **Pop-out windows.** Trading dashboards and monitoring tools that let a user tear a panel into its own window. The panel is adopted into the new document rather than re-created, so live state survives.
- **Print and export via iframe.** Generating an invoice by moving a live DOM subtree into a hidden iframe and calling `print()`.
- **Cross-document drag and drop**, where content moves between same-origin frames.

The mechanism is `document.adoptNode()`:

```
const iframeDoc = iframe.contentDocument;
iframeDoc.body.appendChild(iframeDoc.adoptNode(cardElement));
// adoptedCallback fires on cardElement
```

What belongs in `adoptedCallback` is anything bound to the *old* document: a listener on `document`, a reference to `window`, a stylesheet adopted from the previous context, a `matchMedia` query. Those bindings are now pointing at the wrong document, and only this callback tells you it happened.

You can always ask an element where it currently lives:

```
console.log(this.ownerDocument === document); // false, once adopted
```

2.7 Upgrade timing

Now back to the bug this chapter opened with.

When the HTML parser meets `<my-card>` and no `my-card` is registered, it does not fail. It creates an ordinary `HTMLElement` — a real node, in the DOM, with none of your behavior. This is an *undefined* custom element.

Later, when `customElements.define('my-card', Card)` runs, the browser walks the document, finds every matching undefined element, and **upgrades** each one. It re-runs the element through your constructor, fires `attributeChangedCallback` for observed attributes, and fires `connectedCallback` if it's connected. From that moment the element has your class's prototype and behaves as intended.

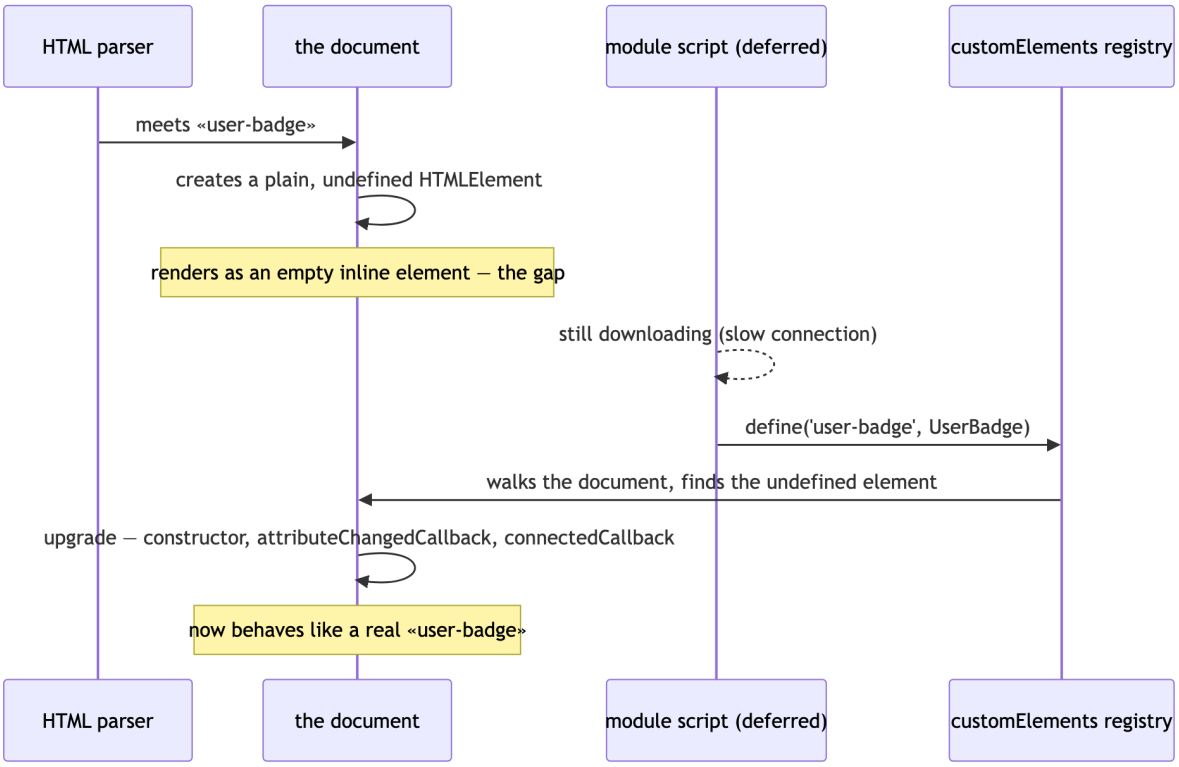
So there are two orderings, and both are normal:

Defined before parsing — a synchronous `<script>` in `<head>`, or any script that runs before the element appears. Elements are upgraded as the parser creates them. No gap.

Defined after parsing — a `<script type="module">`, a deferred script, a dynamic `import()`, anything at the end of `<body>`. Elements exist in the DOM first and get upgraded when the definition arrives. There is a real window during which they render as empty inline elements.

That window is the slow-connection users' entire experience of the component. It isn't an error state. It's the specified behavior of a perfectly correct component. And note that `type="module"` is deferred by default, which means **the modern, recommended way to load a component is also the way that guarantees the gap exists.**

Figure 2.1 — this chapter's cold-open bug, as a timeline:



Nothing above is a race that sometimes goes wrong. It's the specified sequence, every time a script is deferred. The only variable is *how long* the gap between the first two steps and the last three actually lasts. That's exactly the variable a fast development machine and a slow connection disagree about.

You have two ways to deal with it: close the gap, or style it. Closing it means Declarative Shadow DOM, which renders real markup that the parser can paint before any JavaScript arrives — the right answer for content that matters. Styling it is the answer for everything else, covered next.

2.8 Detecting upgraded elements and preventing flash

`:defined` matches elements that have been defined — all built-in elements, always, plus any custom element whose definition has landed. Its negation, `:not(:defined)`, is therefore a precise selector for "custom elements still waiting."

The bluntest version:

```
my-card:not(:defined) {
  visibility: hidden;
}
```

The element still occupies no space and paints nothing, but you've replaced a half-rendered flash with a clean absence. This next version is better, because it acknowledges the element's eventual size and avoids layout shift:

```
my-card:not(:defined) {
  display: block;
  min-height: 120px;
}
```

And the version I reach for most, which turns the upgrade into something that reads as intentional:

```
my-card {
  opacity: 0;
  transform: translateY(10px);
  transition: opacity 0.6s ease, transform 0.6s ease;
}

my-card:defined {
  opacity: 1;
  transform: translateY(0);
}
```

A component that fades in looks designed. The same component appearing abruptly mid-paint looks broken. The DOM behavior is identical.

One caveat that catches people: because `:defined` matches every built-in element, `:not(:defined)` is only ever meaningful on custom element selectors. A blanket `*:not(:defined) { visibility: hidden }` does what you want, but it's fragile. It hides typos in tag names rather than surfacing them. Name the elements you mean.

2.9 Hands-on: `<my-card>`

Let's put the chapter together. This is the component the next several chapters build on, so it's worth typing rather than skimming.

```
// component/card.component.js
class Card extends HTMLElement {
  static observedAttributes = ['message'];

  constructor() {
    super();
    // Shadow root here, so it exists before any callback needs it (§2.5)
    this.attachShadow({ mode: 'open' });
```

```
    this._onClick = this._onClick.bind(this);
  }

  connectedCallback() {
    this.render();
    this.addEventListener('click', this._onClick);
  }

  disconnectedCallback() {
    // Whatever connectedCallback started, stop (§2.4)
    this.removeEventListener('click', this._onClick);
  }

  attributeChangedCallback(name, oldValue, newValue) {
    if (oldValue === newValue) return;
    this.render();
  }

  _onClick() {
    console.log('card clicked');
  }

  render() {
    const message = this.getAttribute('message') ?? 'Default message';
    this.shadowRoot.innerHTML = `
      <style>
        :host {
          display: inline-block;
          border: 1px solid #ccc;
          border-radius: 8px;
          padding: 20px;
          box-shadow: 0 2px 5px rgba(0, 0, 0, 0.1);
        }
      </style>
      <div>${message}</div>
    `;
  }
}

customElements.define('my-card', Card);
```

```
<!-- index.html -->
<link rel="stylesheet" href="style.css">
<script type="module" src="component/card.component.js"></script>

<my-card message="Hello from a web component!"></my-card>
```

```
/* style.css */
my-card {
  opacity: 0;
  transition: opacity 0.4s ease;
}

my-card:defined {
  opacity: 1;
}
```

In a terminal, `cd` into this folder and run `python3 -m http.server`, then open `localhost:8000` in a browser — module scripts won't load from a `file://` URL. Open DevTools alongside it, and try these in order:

1. **Watch the upgrade gap.** In DevTools, open the Network panel and set throttling to "Slow 3G" (the dropdown that normally reads "No throttling"). Reload the page. You should see the card fade in noticeably late, well after the rest of the page — the `:defined` transition in `style.css` is making the undefined-element window visible instead of invisible. Set throttling back to "No throttling" before continuing.

2. **Change the attribute and watch it re-render.** In the DevTools console, run:

   ```
   document.querySelector('my-card').setAttribute('message', 'Changed');
   ```

   The card's text should update to "Changed" immediately — `attributeChangedCallback` fired and called `render()`.

Figure 2.2 — the built card, right after that attribute change: `attributeChangedCallback` fired, `render()` rebuilt the shadow root, and the text now reads "Changed."

3. **Prove `connectedCallback` and `disconnectedCallback` both fire when an element moves.** In the Elements panel, expand the `<my-card>` element's `#shadow-root` and double-click into the `<div>` inside it to edit its text directly in the DOM tree; change it to `MUTATED` and press Enter. Then, back in the console, run:

   ```
   document.body.appendChild(document.querySelector('my-card'));
   ```

   Look at the shadow root's `<div>` again (still in the Elements panel): it should read "Hello from a web component!" again, not "MUTATED". `appendChild` on a node already in the document removes it and reinserts it, which fires `disconnectedCallback` then `connectedCallback` — and `connectedCallback` calls `render()`, which rebuilds the shadow root from scratch and discards your manual edit.

4. **Break it deliberately, to see the ordering rule enforced.** In `component/card.component.js`, cut the line `this.attachShadow({ mode: 'open' });` out of the constructor and paste it as the first line of `connectedCallback`.

   1. Save the file and reload the page in the browser.

2. Look at the console: you should see an uncaught error there — `Cannot set properties of null (setting 'innerHTML')`, thrown from inside `render()`. `attributeChangedCallback` fires *before* `connectedCallback` on upgrade, so `render()` runs and tries to write to `this.shadowRoot.innerHTML` while `this.shadowRoot` is still `null`. The shadow root hasn't been attached yet, because you moved that call downstream of the callback that needs it first.
3. Undo the change: move `this.attachShadow(...)` back to the top of the constructor and save. Do this before moving on, since later chapters build on this file as written above.

That `render()` method rewrites the entire shadow root on every attribute change, which is wasteful and destroys anything stateful inside — focus, scroll position, a half-typed input. I'm leaving it that way on purpose. It's the simplest thing that works, and it's what most hand-written components start as. Chapter 4 retires it, and what replaces it is hand-written bookkeeping that grows with every piece of state. That's what accumulates from there.

What to remember

- **The registry is global, permanent, and first-writer-wins.** `define()` can't be undone or repeated.
- **The constructor may not touch attributes or children.** Attach the shadow root there and nothing else.
- **`connectedCallback` fires on every insertion, not once.** Pair everything it starts with a stop in `disconnectedCallback`.
- **`attributeChangedCallback` fires *before* `connectedCallback` on upgrade**, and fires on every set, not every change.
- **Undefined elements are real, empty, and in the DOM.** That window is normal — close it with Declarative Shadow DOM (§3.8) or style it with `:defined`.
- **Use autonomous custom elements.** Customized built-ins aren't supported in Safari and won't be.