Web Components with Lit

A hands-on guide to building web components and design systems

Sandeep Patel

Web Components with Lit *A hands-on guide to building web components and design systems*

First edition, 2026.

The book targets **Lit 3.3.3**. Specific companion-package versions are listed at their point of use.

Cover and interior design by the author.

Contents

Preface

Who this book is for

This book is for anyone who wants to build user-interface components that outlive the framework they were written in.

That covers a lot of people. You might have never touched a custom element and want to learn the platform from the ground up. You might be fluent in React or Vue or Angular and be curious why anyone would reach for the browser's own component model instead. You might already ship Lit and want to get deliberate about controllers, accessibility, testing, and packaging. All three of you are welcome here, and the book is built so that none of you has to read the parts you already know.

What the book *does* assume: you can write modern JavaScript (arrow functions, modules, classes, `async/await`, destructuring), and you have used the DOM directly at least once, even if you didn't enjoy it. You do not need TypeScript. You do not need a build tool. You do not need to have built a component library before; we build one together, from the first `customElements.define` to a package published on npm.

The four reader paths

The chapters are ordered so the book reads straight through, but four shortcuts through it are worth naming.

New to web components. Read Part I in full: it teaches custom elements and shadow DOM with no framework at all, because Lit is a thin layer over those and the layer makes more sense once you have felt the bare platform. Then keep going in order.

Coming from a framework. Skim Part I for the parts of the platform your framework hides from you (the upgrade problem, the shadow boundary, retargeted events), then slow down at Part III, where Lit's reactivity model diverges from what you are used to. Chapter 8, on how Lit renders, is worth reading early.

Already using Lit. Start at Part V. Controllers, context, state, and forms are where most Lit codebases accumulate their worst patterns, and Parts VI and VII (the design system, accessibility, testing, SSR, publishing) are the material that is hardest to find written down anywhere else.

Here for one topic. The chapters are short and single-purpose. If you opened this book because your custom element won't submit inside a `<form>`, go straight to Chapter 23 and come back later. The page after this preface, "If You Came Here with a Problem," maps around fifty of the most common symptoms (`checked="false"` is still checked, `shadowRoot` is `null`, the event never fires) to the section that explains each one.

Throughout the book, sidebars headed **New to web components**, **Coming from a framework**, or **Already using Lit** flag places where that audience can skip ahead or should slow down. The heading is the label: no icon or color to keep track of, in any edition.

JavaScript first, TypeScript alongside

Every example in this book is plain JavaScript. Lit works with no compiler, and the standards-only syntax (`static properties`, no decorators) is the version that will still run unchanged in five years. Where TypeScript changes the picture, a short `> TS` callout shows the decorator-based equivalent, and Appendix A collects all of it in one place. If you write TypeScript, you lose nothing by reading the JavaScript; if you don't, you are never blocked on a `tsconfig`.

The case study

One project runs through the book. **Cinder** is a small design system: a real component library, published on npm as `cinder-ui`. It starts in Chapter 6 as a button that does almost nothing and ends in Chapter 36 as a versioned, documented, server-renderable package on the npm registry, with a React wrapper package and written guidance for using it from any framework. By the end you will have built a button, an icon, a text field and checkbox that work inside native forms, an accessible dialog, a toast system, and a theming layer.

The examples that consume Cinder are plain HTML pages, one per idea. There is no demo application to build and no router to wire up: a page imports the components it needs and shows them working. The few chapters that need more than one page, on routing and server-side rendering, build a small set for the occasion.

Cinder is not a toy and not a second framework. Every component earns its place by teaching something a chapter needs. That is also why it stays small: eleven components (button, icon, spinner, a field wrapper and the three form controls that use it, dialog, tooltip, toast, and a theme provider), not the fifty a real design system eventually grows. There is no data table, no menu, no date picker, and Chapter 24 argues directly against building one: a component that tries to own sorting, filtering, pagination, and editing all at once is a framework wearing a mask, and the right fix is usually several small components instead of one that does everything. Cinder is deliberately the size a book can build and a reader can hold in their head, not a competitor to a production design system.

It is also real, not a prop: `cinder-ui` and `cinder-react` are published on npm under those exact names and install with `npm install cinder-ui`. A fresh project installing it from the registry (not a local build) was checked once the package went live: the components register, `internal/` stays blocked, and Lit is deduplicated to a single copy, exactly as Chapter 35 describes.

A note on how the code was checked

Every runnable listing in this book was actually run (in a real browser or a real terminal) before the surrounding text described what it does. That includes the listings that are supposed to fail: those were run too, to confirm they fail the way the text claims. Every cross-reference was checked against its real target. Every version number and "at the time of writing" was verified against the live source, not remembered.

This matters more for a programming book than almost any other kind of writing, because you are going to run this code, and folklore about how a platform behaves goes stale fast. Where checking the code caught the first draft being wrong about something, the correction was left visible in the text rather

than quietly edited away: those moments are usually more instructive than the parts that were right the first time.

The version this book targets is **Lit 3.3.3**, the current release as of writing. Lit follows semantic versioning and the 3.x line has been stable; where a companion package is still pre-1.0 (`@lit/localize` and everything under `@lit-labs/`), the text says so at the point of use.

If You Came Here with a Problem

Most people open a technical book with a question, not a syllabus. This page maps the questions that bring developers to Lit and web components (the ones that fill search results and issue trackers) to the section of this book that answers each. Every entry was a real bug at some point during the writing; several were the author's.

The element itself

Symptom	Where
I set `checked="false"` (or `disabled="false"`) and it is still on	Ch. 3, 9, 11
I set a property on my element and nothing re-rendered	Ch. 1, 3
`this.shadowRoot` is `null` in the constructor or `connectedCallback`	Ch. 10
My element was in the HTML but had no behavior until later	Ch. 2
A timer or listener keeps running after the element is removed	Ch. 2, 14, 19
Moving an element in the DOM fired `connectedCallback` again	Ch. 2
Infinite update loop, or "scheduled an update after an update completed"	Ch. 10, 30
I overrode `updated` / `connectedCallback` and things stopped working	Ch. 10

Templates and rendering

Symptom	Where
The template shows the text "0", "false", or "NaN"	Ch. 11
The attribute is there but empty instead of absent	Ch. 11
`<input .value=${x}>` won't update to a value it already had	Ch. 13
After sorting a list, a checkbox / focus ended up on the wrong row	Ch. 13
Toggling a class rebuilt the subtree and lost focus	Ch. 8
`classMap` throws about the attribute value	Ch. 13
I can't build a template from a string	Ch. 8, 12
`unsafeHTML`: is this an XSS hole?	Ch. 12
Bare `import 'lit'` fails in the browser with no bundler	Ch. 7
My directive leaks timers / stops working after a `repeat` reorder	Ch. 14

Styling and theming

Symptom	Where
The component renders in the page's font or color	Ch. 4, 15
The page's stylesheet can't reach inside my component	Ch. 4, 16
A subclass lost the base class's styles	Ch. 15
Dark mode: components didn't follow the theme switch	Ch. 17
A `::part` on a nested component can't be reached from the page	Ch. 16
`:host-context` doesn't work in Firefox	Ch. 16

Events and communication

Symptom	Where
My custom event never reaches a listener outside the component	Ch. 4, 18
`event.target` is the wrong element	Ch. 4, 18
Should this be an `onChange` prop or an event?	Ch. 18, 24
Two sibling components need the same state	Ch. 18, 21, 22

Data, state, and async

Symptom	Where
Fetching in `connectedCallback` shows the wrong user after a fast id change	Ch. 20
A context consumer gets `undefined`	Ch. 21
The store changed but subscribed components didn't re-render	Ch. 22
Do I need Redux / a store at all?	Ch. 22

Forms and accessibility

Symptom	Where
My element's value isn't in `FormData`; `required` doesn't block submit	Ch. 23
`form.reset()` doesn't reset my element	Ch. 23
Clicking the `<label>` doesn't focus my element	Ch. 23
`<label for>` / `aria-describedby` doesn't work across the shadow root	Ch. 25
`axe` says "Form elements must have labels" and I have a label	Ch. 25
Screen reader doesn't announce the toast / the error	Ch. 25
The dialog doesn't trap focus or give it back	Ch. 26
`await animation.finished` never resolves	Ch. 27

Delivery

Symptom	Where
The route works until the first hard refresh	Ch. 29
Focus and scroll don't move on client-side navigation	Ch. 29
`import 'cinder-ui'` did nothing in my bundled app	Ch. 32, 35
The consumer's bundle contains two copies of Lit	Ch. 35
`window is not defined` during server render	Ch. 34
Hydration re-rendered everything / a flash on load	Ch. 34
`cem analyze` produced nothing or garbage	Ch. 28; App. D
The visual test fails on the second run with nothing changed	Ch. 31

Frameworks and TypeScript

Symptom	Where
React doesn't fire my `cn-change` handler	Ch. 36
Vue warns "Failed to resolve component: cn-…"	Ch. 36
Angular: "cn-button is not a known element"	Ch. 36
A reactive property stopped updating after I enabled decorators	App. A
`TS1240` on a `@property()` decorator	App. A
Does this feature work in Firefox / Safari yet?	App. B

How to Use the Example Code

Every runnable listing in this book comes from a companion code bundle, extracted mechanically from the chapter source so the printed code and the bundle cannot drift apart. You do not need the bundle to read the book, but you will get more out of it if you run the examples as you go.

Getting it

Download the archive:

https://drive.google.com/file/d/1srfXNb0AdfDKo-6ygkeTJxM7ff_X9684/view?usp=sharing

Unzip it and change into the folder:

```
unzip web-components-with-lit-code.zip
cd web-components-with-lit-code
```

The download link is also on the book's page at the store you bought it from.

You need **Node 20 or newer** and a modern browser. That is all for most of the book.

The layout

```
examples/          one folder per chapter: examples/ch04/encapsulation/, …
packages/cinder/   the Cinder design system, built up across the book
packages/cinder-react/   the React wrapper package from Chapter 36
e2e/               the end-to-end test from Chapter 31
SOLUTIONS.md       answers to every "Try it" exercise
```

Each example folder under `examples/chNN/` is a self-contained page for one idea. Most are a single `index.html` that imports what it needs: nothing to install.

Running the pages

The example pages load Cinder and the theme tokens from `/packages` through an import map, so they only work when the server's root is the top of the unzipped folder, not an individual example. Start the included static server:

```
npm run serve
```

Then open the page for whatever you are reading, for example:

```
http://localhost:8000/examples/ch01/hello/
http://localhost:8000/examples/ch18/filter-bar/
```

A handful of examples (the localization workflow in Chapter 33, the server-rendered page in Chapter 34) have their own `package.json` and a short README in the folder, because they need a build step or a Node server of their own.

The Cinder library

The case-study library lives in `packages/cinder/`. To build it, lint it, type-check its templates, and run its test suite in one command:

```
cd packages/cinder
npm install
npm run check
```

The individual scripts (`npm run build`, `npm test`, `npm run test:visual`, `npm run analyze`) are introduced in the chapters that set them up (7, 28, 30, 31, 35).

The exercises

Every chapter ends with two short **Try it** exercises. They are deliberately small: change one line, run one command, read what comes back, and most of them are set up so the result is not what you would guess. Do them before you look anything up.

`SOLUTIONS.md` at the top of the bundle has an answer for each one: what you should see, why, and the section to reread if it surprises you.

When the code and the book disagree

The book was written against fixed versions: Lit 3.3.3, and the companion-package versions named at each point of use. If you install newer versions and something behaves differently, the bundle's `package.json` files pin what the book actually used; start there. And where running the code caught the first draft of the book being wrong about something, the correction was left in the text on purpose, so a passage that reads like the author changed their mind mid-paragraph is doing exactly that.

Acknowledgements

This book stands on work done in the open by other people.

The Lit team at Google built the library the book is about, and (just as importantly) wrote and maintained the documentation, the RFCs, and the years of design discussion that made it possible to explain *why* Lit works the way it does, not just how. The `@lit-labs` packages, still moving fast, are the reason chapters on SSR, localization, and context could be written at all.

The web-components community that predates Lit: the people who fought for Custom Elements and Shadow DOM to ship as standards, who wrote the polyfills that carried the idea through the years before the browsers caught up, and who built the tooling this book leans on: `@web/test-runner`, `@open-wc/testing`, `@custom-elements-manifest/analyzer`, `lit-analyzer`, the ESLint plugins. Open a `node_modules` folder for any chapter of this book and you are looking at a decade of donated work.

The "verify, don't assert" discipline the book is built on came from writing a previous technical book and getting caught, more than once, having trusted a memory of how something behaved instead of checking. Those corrections were the best teachers.

Any errors that remain are the author's, and corrections are welcome at the address on the book's store page.

About this sample

This is a free sample of *Web Components with Lit* — the front matter, Chapter 1 (the case for the platform, and a first Lit component with no build step), and Chapter 6 (the first full Lit component, and the start of the case study). Chapters 2 to 5, between them, build the same things by hand so the library makes sense when it arrives. The full book is 36 chapters and five appendices, and carries one project — the **Cinder** design system — from a first `customElements.define` to a package published on npm.

PART I

Web Components without a Framework

Chapter 1: Why Web Components, Why Lit

Front-end frameworks have a shelf life of about five years. If you have been building for the web for any length of time, you have rebuilt the same dropdown more than once, in more than one framework, and you did not enjoy it.

Web Components are the browser's answer to that. You build a component once, as a real HTML element, and it works in React, in Vue, in a plain page, and in whatever replaces all of those. Lit is a small library that makes those components pleasant to write instead of tedious.

This chapter is the pitch. There is nothing to install; the one example at the end runs straight from a CDN.

This chapter covers:

- The interoperability problem: why a component built for one framework can't be used by another, and what that costs a team.
- The four browser features that "Web Components" refers to: custom elements, Shadow DOM, `<template>`, and ES modules.
- How the browser turns a plain tag into an instance of your class.
- What Lit adds, and the four things it leaves out on purpose.
- When Lit is the wrong tool.
- A complete Lit component running with no build step.

1.1 The interoperability problem

Picture a product with three front ends, which is a normal situation and not a made-up one. There is a marketing site built with a static site generator. There is the web app, a React single-page app. And there is a documentation portal running on whatever the docs team chose years ago. All three need the same Sign up button: same padding, same focus ring, same spinner while it's working.

You cannot hand a React component to the two front ends that aren't React. So the button gets built three times, and the three copies are kept in sync by hand and good intentions. That holds up until the day someone fixes the focus ring in one of them and forgets the other two.

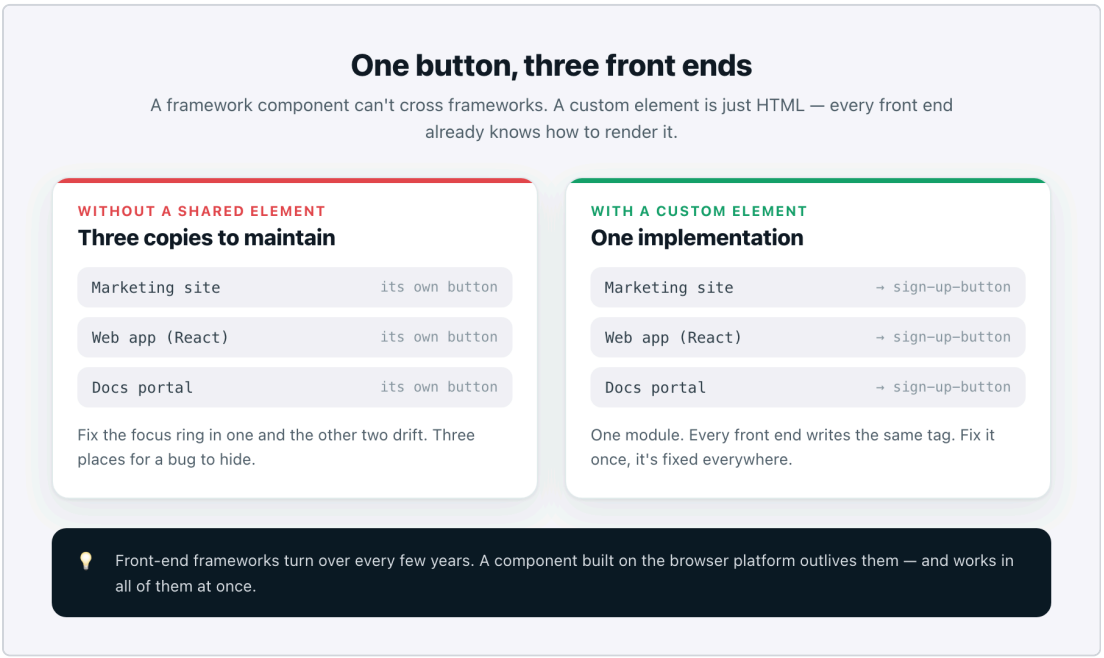

Figure 1.1: One button, three front ends. Kept as three copies, they drift apart. Built as a custom element, there is one implementation and all three render it.

Standardizing on one framework everywhere would fix the drift, but that is a year of migration work that ships no features. A custom element fixes it a different way. `<sign-up-button>` is a tag you teach the browser about once. Every front end loads the same module and writes the same tag, because every front end is producing HTML for the same browser in the end. Fix the button in that one module and it is fixed in all three places at once.

> **Analogy.** A framework component is furniture bolted to the wall. A custom element is a chair: you can carry it to a different house and it still works, because every house has a floor.

This is why design-system teams at large companies ship their libraries as custom elements. Product teams on different frameworks, and product teams that will change frameworks in two years, all consume the same components without a rewrite.

1.2 What “Web Components” means

“Web Components” is not one thing. It is four browser features that happen to work well together. None of them is a library, and all four ship in every current browser.

Feature	What it does
Custom elements	Registers a JavaScript class as the implementation of a tag, with lifecycle callbacks.
Shadow DOM	Gives an element a private DOM subtree the page can't style into or query by accident.
HTML `<template>`	Holds inert markup you stamp out copies of: the basis of efficient rendering.
ES modules	Native `import` / `export`, so a component loads without a bundler.

Between them, they let you ship an element that carries its own markup, styles, and behavior and drops into a page as one tag.

The feature that makes a custom element feel like a built-in one is the lifecycle. When the browser parses `<sign-up-button>Sign up</sign-up-button>`, this is what happens:

1. It sees a hyphenated name it doesn't recognize and treats it as a custom element.
2. If a class has been registered for that name with `customElements.define('sign-up-button', …)`, the browser *upgrades* the element: it runs the constructor and turns the plain tag into an instance of your class.
3. `connectedCallback()` fires. This is where you render, attach listeners, start timers.
4. When the element leaves the page, `disconnectedCallback()` fires, and you undo all of that.

If the class is registered after the tag was already parsed, the upgrade just happens then instead. You don't have to worry about load order.

Chapter 2 builds a working custom element on these APIs with no library, so you can see exactly what Lit is doing for you later.

1.3 Why the raw platform isn't enough

So why isn't that the whole story? Because the four features are deliberately low-level. They hand you the pieces to build a component, but two jobs that every component needs are left for you to write, and you end up writing them again in each one.

The first is updating the DOM. The platform is good at *creating* nodes (`<template>`, `cloneNode`, `innerHTML`) and says nothing about *changing* them once they exist. When a piece of your component's data changes, you have two options. You can rebuild the whole subtree from scratch, which is wasteful and also throws away everything the browser was quietly keeping track of: the caret position in a text field, the user's selection, an animation partway through, the scroll offset of a long list. Or you can write code that works out exactly which text node, attribute, or child element goes with the value that changed, and touches only that. For a list, done properly, that second option is a small reconciliation algorithm: match the old items to the new ones, work out what moved, what was inserted, what was removed. You would write a version of it in every component that renders a collection. Chapter 5 builds it by hand and shows where it stops being worth doing.

The second is noticing that the data changed at all. Set a property on a plain custom element: `el.label = 'Saved'`, and nothing happens. The element does not re-render, because nothing told it to. To make that work you write an accessor that runs on every assignment, checks whether the value changed at all, and

schedules a render if it did. *Schedules*, not runs: if three properties change in the same click handler you want one render at the end, not three along the way, so the accessor has to defer the work and coalesce it. And the first render has to land at the right moment in the element's life, after it is connected but before the reader sees a flash of empty component. Chapter 3 builds this for a single property and counts the moving parts; there are six or seven, and that is *per property*.

Neither job is conceptually hard. Both are boilerplate, both are near-identical from one component to the next, and boilerplate copied across forty components is where the subtle bugs settle in: a missing change check that spins into an infinite render loop, a teardown you forgot that leaks a listener every time the element mounts. This is the exact code Lit writes once, the same way for every component, so that you never write it at all.

1.4 What Lit adds, and what it doesn't

Lit is a small library (about 6 kB minified and gzipped) that sits on the four platform features and does those two chores for you.

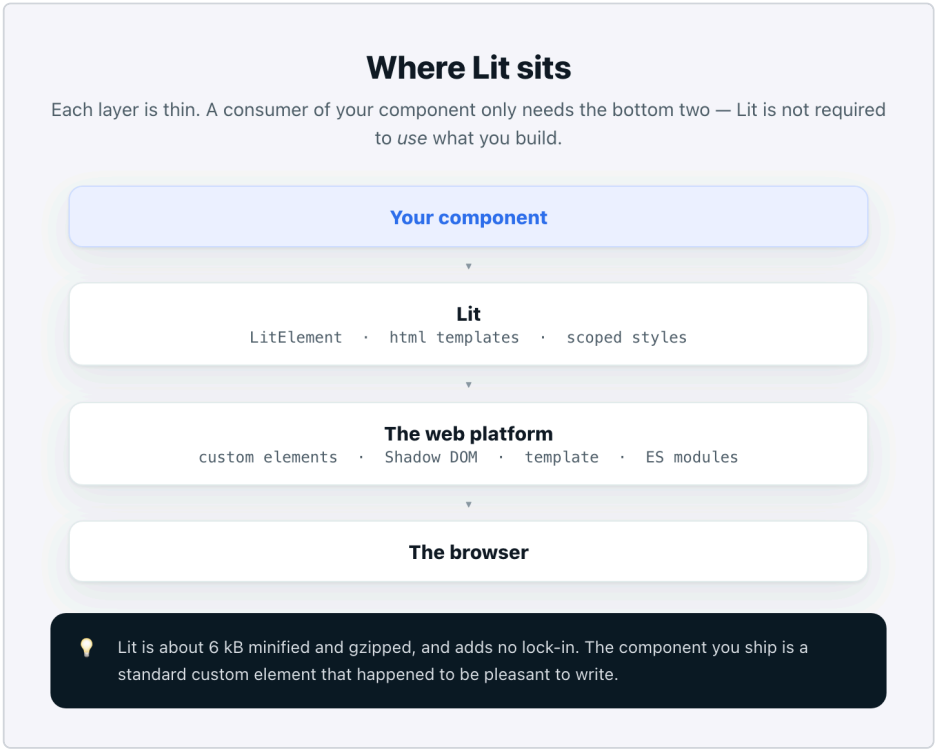

Figure 1.2: Where Lit sits. Your component uses Lit, Lit uses the platform, the platform is the browser. Anyone consuming your component only needs the bottom two layers.

There are three parts to it:

- **`LitElement`**, a base class that runs the update cycle. Declare a property, change it later, and the element re-renders, with several changes in one tick batched into a single update.
- **`html` templates**. You write markup as a tagged template literal, and Lit re-evaluates only the dynamic bits on each update.
- **Scoped styles**, parsed once per component class and shared across every instance.

That is close to the whole library.

It also leaves a lot out, and that is deliberate:

1. **No router.** A component is an element; routing is an application's job (Chapter 29).
2. **No global state management.** You get reactive properties and a way to pass data down a tree with context. Anything bigger is a choice you make (Chapter 22).
3. **No build step.** Lit runs in the browser as shipped. A bundler is useful for a large app, but it is never required, and the examples in this book prove it.
4. **No application layer.** No `App` component, no page lifecycle, no opinion about how your project is laid out.

The small surface is the point. What you build with Lit is an ordinary custom element that was pleasant to write, and nothing downstream of it needs to know Lit was involved.

> **Analogy.** Lit is power steering, not a self-driving car. The browser is still doing the driving. Lit takes the effort out of the turning, and leaves the decisions to you.

One cycle sits under everything Lit does: how a change to a property turns into a change in the DOM.

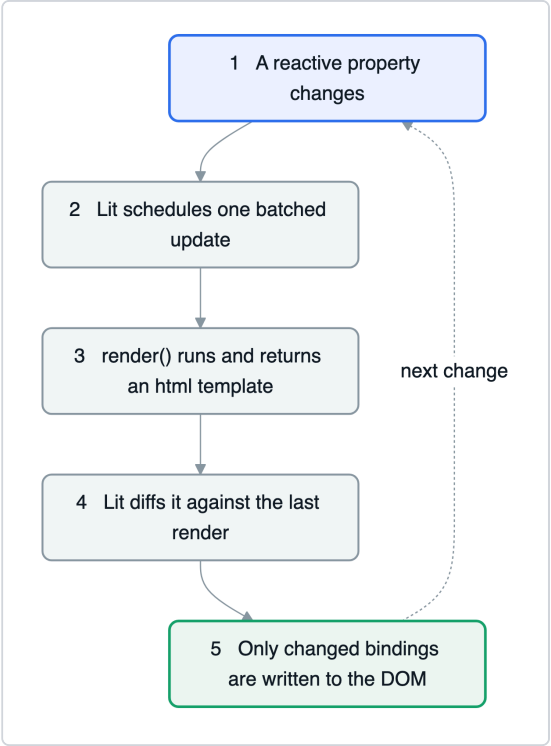

Figure 1.3: Lit's reactive update cycle, from a property change (step 1) to a minimal DOM write (step 5), then again on the next change.

You never call `render()` yourself. Changing a reactive property is the only thing that starts the cycle.

Two details in that cycle are what keep updates cheap. First, changes are grouped: if you set three properties one after another, Lit still renders once, not three times. Second, a render does not rebuild the element. Your template is mostly fixed text with a few slots for data, and Lit only ever revisits those slots, updating the ones whose value changed. A typical render comes down to rewriting a single piece of text. Chapter 8 gets into how; here it is enough that it happens on its own.

1.5 A Lit component is a real HTML tag

A framework component only exists inside its framework. A React `<UserCard>` means nothing to the browser; the browser only ever sees the DOM that React produced. You can't take that component and drop it into a Vue app or a plain page.

A Lit component is not like that. This class:

```
import { LitElement, html } from 'lit';

class UserCard extends LitElement {
  static properties = { name: {}, role: {} };

  render() {
    return html`
      <div class="card">
        <h2>${this.name}</h2>
        <p>${this.role}</p>
      </div>
    `;
  }
}
customElements.define('user-card', UserCard);
```

registers `<user-card>` as a real tag. Once its module has loaded, this line works anywhere, in a framework app or a static HTML file:

```
<user-card name="Ada Lovelace" role="Mathematician"></user-card>
```

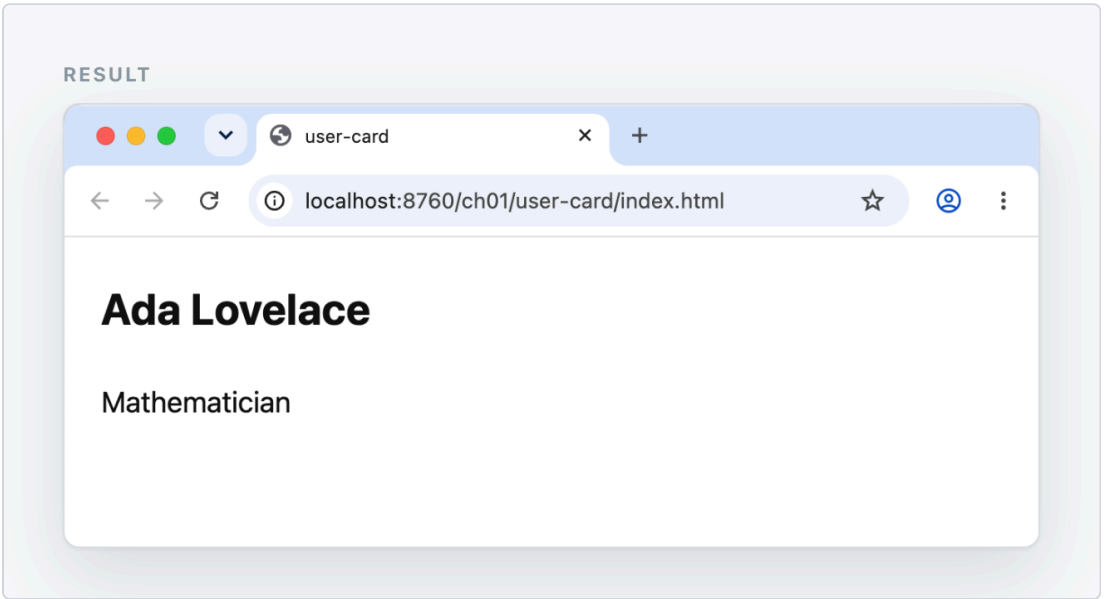

Figure 1.4: What `<user-card>` renders: a plain `<h2>` and `<p>` in the browser's default type. Making it look designed is the job of Chapters 15–17.

What each line does:

- `import { LitElement, html } from 'lit'` brings in the base class every Lit component extends, and the `html` tag you write templates with.
- `class UserCard extends LitElement`: a component is a class, and extending `LitElement` is what wires it into the update cycle from Figure 1.3.
- `static properties = { name: {}, role: {} }` declares `name` and `role` as *reactive* properties: change either one and the component re-renders on its own. The empty `{}` is that property's options object, left at the defaults for now; Chapter 9 is about what goes in it.
- `render()` returns the markup, written with the `html` tag, which is a tagged template literal. Lit reads this template once and remembers where the dynamic positions are.
- `${this.name}` and `${this.role}` are *bindings*. Each render, Lit checks whether the value changed and updates just that piece of text if it did, leaving the `<div>` alone.
- `customElements.define('user-card', UserCard)` connects the class to the tag. After it runs, every `<user-card>` on the page is a `UserCard`. The name has to contain a hyphen; that's how the browser tells your elements from its own.

No framework runtime, no build step, no template compiler. The browser gets a tag it knows how to build. Chapter 36 covers what "works anywhere" means in practice for each framework.

1.6 When Lit is the wrong choice

Lit is not always the right call.

If you are building one large application with a lot of client state and routing, and none of it needs to be reused anywhere else, a full framework hands you a router, data-fetching conventions, and app-shaped tooling for free. You can build a whole app out of Lit components, and people do, but you are signing up to make some decisions yourself that a framework would otherwise make for you.

And if your team is on one framework, plans to stay there, and ships nothing that other teams consume, the interoperability argument doesn't apply to you. Framework-native components will fit the rest of your code better.

The clear case for Lit is a component library or design system that has to cross frameworks or outlast one. It also suits smaller jobs: interactive pieces on pages that aren't single-page apps, or anywhere you want real style encapsulation and a small dependency footprint.

1.7 How to read this book

The preface has the full roadmap: four reader paths and a symptom index for anyone who arrived with one specific bug. Whichever path you take, the code conventions are the same throughout: every listing is plain JavaScript, a `> TS` callout marks the few places TypeScript changes the picture, and Appendix A collects the TypeScript version of the whole book in one place.

1.8 Setup

You need three things.

1. **Node.js**, a current LTS or newer, for the project examples and the build tooling from Chapter 7 on. Check it with `node --version`:

   ```
   $ node --version
   v22.14.0
   ```

   Anything on `v20`, `v22`, or later is fine.
2. **A terminal.** Commands are written for a POSIX shell (macOS, Linux, WSL) and translate directly to PowerShell.
3. **A current browser.** Chromium, Firefox, or Safari. Lit 3 targets modern browsers and does not support Internet Explorer 11 or the old Edge. The examples were checked in current Chrome.

Several of the early chapters need nothing but a browser and an editor.

The code blocks in this book come in three kinds. A block with a `$` prompt, like the one above, is a terminal session; what follows the command is its real output. A block with no prompt is a file or a listing you can copy. And a **Result** figure, like Figure 1.4, shows what a listing renders, so you can see the outcome without running it.

1.9 Your first Lit component

The rest of this chapter has argued for the platform and for Lit as a thin layer over it. The quickest way to see what that layer buys you is to write one component. Lit is a library, not a framework or a compiler, so a single `<script>` tag loading it from a CDN is enough: there is no project to create and no build to run. The component below defines a `<hello-lit>` tag that renders a greeting and re-renders whenever its `name` changes.

Save it as `hello.html` and open it in a browser:

```
<!-- hello.html -->
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Hello Lit</title>
  </head>
  <body>
    <hello-lit name="world"></hello-lit>

    <script type="module">
      import {
        LitElement,
        html,
      } from 'https://cdn.jsdelivr.net/gh/lit/dist@3/all/lit-all.min.js';
```

```
    class HelloLit extends LitElement {
      static properties = { name: {} };

      render() {
        return html`<p>Hello, ${this.name}!</p>`;
      }
    }
    customElements.define('hello-lit', HelloLit);
  </script>
 </body>
</html>
```

You see **Hello, world!**. Now open the browser console and run:

```
document.querySelector('hello-lit').name = 'Ada';
```

The text changes to **Hello, Ada!** right away.

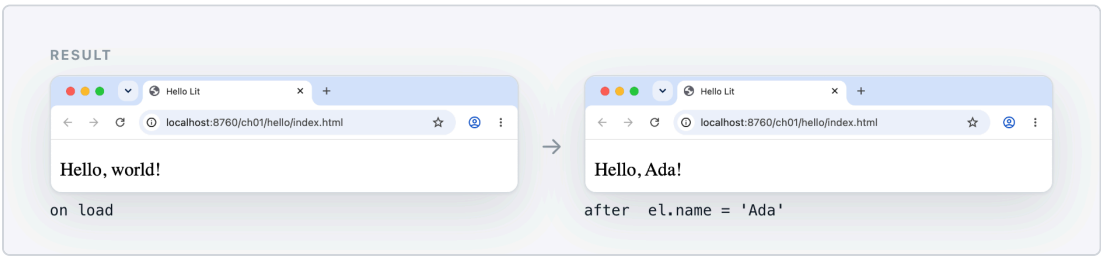

Figure 1.5: `hello.html` before and after the console command. Setting `el.name` re-renders only the text bound to it.

`HelloLit` is the same shape as `UserCard`, with one property instead of two. The parts that are specific to running this in a bare file:

- `<script type="module">`. Lit is distributed as ES modules, and `import` only works inside a module script.
- The import is a full `https://` URL rather than the bare `'lit'` you'd write in a project. A bare name needs a bundler or an import map to resolve; a URL works on its own. This URL points at a prebuilt bundle that includes every built-in directive, so nothing you paste from a later chapter breaks on a missing import. Chapter 7 switches to installing `lit` from npm, where a bundler ships only what you use.
- `<hello-lit name="world">` sets `name` through an *attribute*. Attributes are always strings. `static properties` is what tells Lit to also expose `name` as a real property and keep the two in step (Chapter 9).
- In the console you set the *property* directly, `element.name = 'Ada'`. Either route triggers the cycle in Figure 1.3.

That is the whole idea of the library, in about a dozen lines: you changed a value, and Lit updated the one piece of the page that depended on it, with no build step in sight. Everything else Lit gives you is built on that.

> **Coming from a framework.** If `hello.html` made sense and custom elements, shadow DOM, and slots are already familiar, you can jump from here to Chapter 6 and read Part I only for the two things your framework hides: the upgrade problem (Chapter 2) and event retargeting (Chapter 4).

1.10 The case study: the Cinder design system

From Chapter 6 onward, one piece of code grows alongside the text. **Cinder** (`cinder-ui`) is a design system, an actual component library. It starts as a button that barely does anything and turns, a chapter at a time, into a published npm package: button, icon, spinner, text field, checkbox, select, dialog, toast, a theming layer, a documentation site. By Chapter 35 it is versioned, has a custom-elements manifest, and renders on a server; Chapter 36 adds a React wrapper package. Nothing is added for show. Every component is there because a chapter needs it.

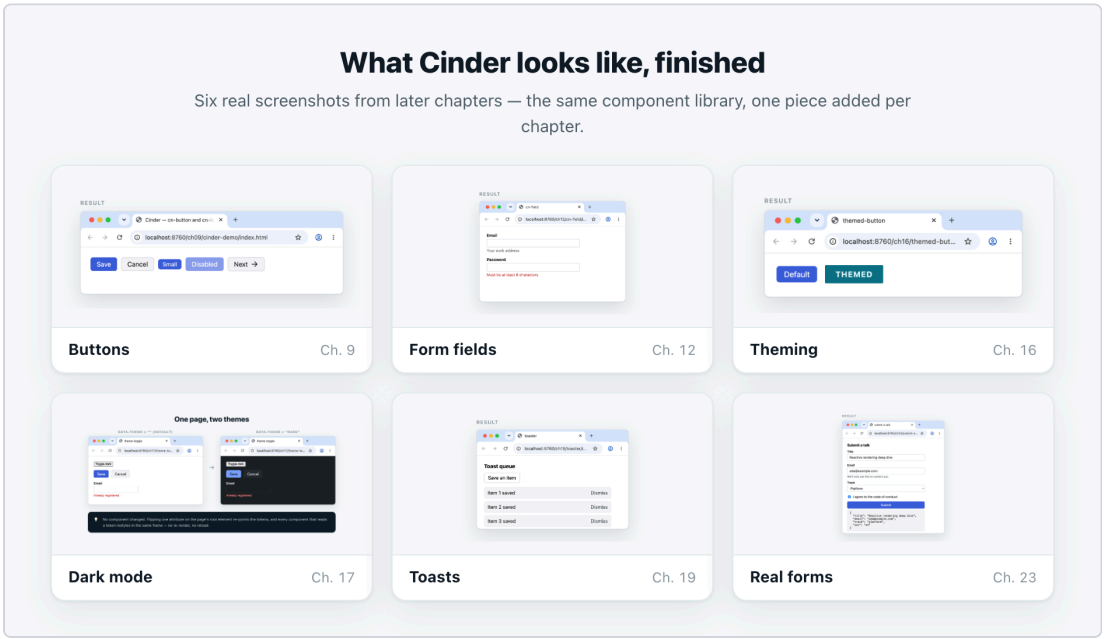

Figure 1.6: What Cinder looks like by the time each of these chapters is done. Every image here is a real screenshot reused from later in the book: nothing here is a mockup.

The code that *uses* Cinder is plain HTML pages, one per idea: a page with a form, a page that fetches data, a page with a theme switcher. None of it is a full application. There is no router, no app shell, and nothing to build before you can open a page in a browser. Two chapters, on routing and server-side rendering, need more than one page, and they build a small set for the occasion.

You don't have to build any of it. Every listing stands on its own. But `examples/` has the code, one folder per chapter, if you want the exact state of it at any point.

Summary

- A component built for a framework only runs in that framework. A custom element is a real HTML tag, so it runs everywhere, including pages with no framework. That is what lets it outlast the framework you're using now.
- "Web Components" means four browser features together: custom elements (a class behind a tag), Shadow DOM (encapsulation), `<template>` (inert, cloneable markup), and ES modules.
- The browser *upgrades* a custom element when its class is registered, running the constructor and firing `connectedCallback()`, whether the definition loads before or after the tag.
- The raw platform makes you write your own rendering and your own reactivity. Lit does both, in about 6 kB minified and gzipped, through `LitElement`, `html` templates, and scoped styles. It has no router, no store, no build step, and no application framework.
- What you build with Lit is a standard custom element; consuming it never requires Lit.
- Reach for a framework instead when you have one large single-framework app with nothing to reuse. Reach for Lit for design systems, cross-framework components, and interactive pages that aren't single-page apps.

Try it

1. Put two `<hello-lit>` elements on the page and set `name` on only one of them from the console. Confirm the other one does not change.
2. Delete the `static properties` line from `hello.html`. Does `name="world"` still render on load? Does `el.name = 'Ada'` still update it?

Chapter 2 builds a custom element with no library, so the boilerplate Lit removes is something you have felt for yourself.

PART II

Getting Started with Lit

Chapter 6: Your First Lit Component

Part I built components by hand: `attachShadow` in the constructor, `observedAttributes`, an `attributeChangedCallback` that calls a `render` method, a getter and setter for every property, a `<template>` you clone and patch. It works, and it is a lot of code that is the same in every component.

Lit replaces all of it with a base class. You extend `LitElement`, declare your properties in one place, write a `render()` method that returns markup, and the update machinery (upgrade, first render, change detection, batched re-render, scoped styles) comes with it.

This chapter builds one small component end to end and traces what happens when it runs. From here on, every component in the book is a Lit component, and the Cinder design system starts at the end of this chapter with its first button.

> **Already using Lit.** Parts II and III are review for you, with two exceptions worth the time: the section on what happens when it runs, later in this chapter, and all of Chapter 8, which give the update model that Parts IV to VII lean on, and Chapter 10's list of the three lifecycle bugs. Then go to Part V.

This chapter covers:

- Extending `LitElement` and registering the tag.
- `render()` and the `html` tagged template.
- Declaring a reactive property with `static properties`.
- Scoped styles with `static styles` and the `css` tag.
- A step-by-step trace from `customElements.define` to a live reactive update.

6.1 Extending `LitElement`

`LitElement` is a subclass of `HTMLElement`. Everything Chapter 2 covered still applies: it is a custom element, registered the same way, with the same lifecycle callbacks. You write a class that extends it, give it a `render()`, and register it.

```
import { LitElement, html } from 'lit';

class GreetingMessage extends LitElement {
  render() {
    return html`<p>Hello there.</p>`;
  }
}

customElements.define('greeting-message', GreetingMessage);
```

On a page, `<greeting-message></greeting-message>` renders `<p>Hello there.</p>` into its shadow root: nothing dynamic yet. The rest of the chapter adds a property and styles to that skeleton.

What the base class adds is a managed *update cycle*: the two things that were tedious by hand in Part I. It attaches a shadow root for you, calls `render()` at the right moments, and puts the result in the shadow DOM; and it makes properties reactive, so changing one re-renders the element on its own. You don't write `attachShadow`, you don't touch `this.shadowRoot`, and you don't call `render()` yourself.

The `import` is a bare specifier, `'lit'`. In this chapter's examples an import map resolves it to a browser build; Chapter 7 installs `lit` from npm and a bundler resolves it instead. Either way the code you write is `from 'lit'`.

> **TS.** With TypeScript and decorators enabled you can write `@customElement('greeting-message')` above the class instead of calling `customElements.define` below it. The two are equivalent; this book uses the standards form in the main text and shows the decorator in Appendix A.

6.2 `render` and the `html` tag

`render()` is where a Lit component says what its shadow DOM should look like, given its current state. Lit calls it for you: after the first connection, and again after any reactive property changes, so you never call it yourself and never touch the DOM directly.

```
render() {
  return html`<p>Hello there.</p>`;
}
```

`html`...`` does not return a string and does not touch the DOM. It returns a `TemplateResult`: a lightweight object holding the template's static strings and its dynamic values. Lit takes that object, and on the first render it builds the DOM once; on later renders it compares the new result against the last one and updates only what changed. Chapter 8 is about how that works. For now: `render()` is a pure description of the output for the current state, and calling it has no side effects.

6.3 A reactive property

A component with no inputs isn't much use. `static properties` declares the properties that, when changed, trigger a re-render.

```
class GreetingMessage extends LitElement {
  static properties = {
    name: {},
  };

  constructor() {
    super();
    this.name = 'there';
  }

  render() {
    return html`<p>Hello, ${this.name}.</p>`;
  }
}
```

Three things are wired up by that one `name: {}` entry:

- Lit generates a getter and setter for `this.name`. The setter compares the new value to the old one and, if it differs, schedules an update.
- `name` is also an *attribute*. `<greeting-message name="Ada">` sets the property to `"Ada"` at startup.
- `${this.name}` in the template is a binding. When `name` changes, Lit re-runs `render()` and writes the new text into that one spot, leaving the `<p>` element itself in place.

The constructor sets a default. `static properties` declares that `name` exists and is reactive; it doesn't give it a value, so without the constructor line `this.name` is `undefined` until an attribute or assignment sets it. Chapter 9 covers the options that go inside the `{}` (type conversion, attribute naming, reflection), all of which you built by hand in Chapter 3.

Compare this to the `<name-tag>` from Chapter 3, which did roughly the same thing: a private field, a getter, a setter with two guards, an `attributeChangedCallback`, an initializer in `connectedCallback`, and a `render` call from three places: six or seven pieces of bookkeeping per property, all of which Lit's one line replaces.

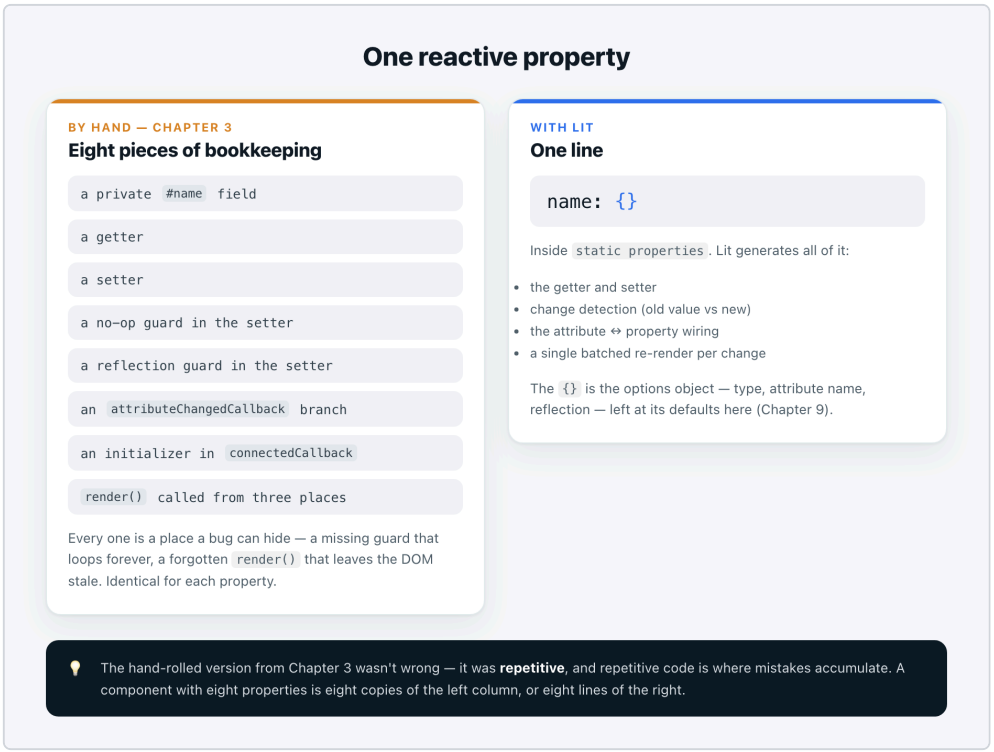

Figure 6.1: The same reactive property, by hand and with `static properties`.

6.4 Scoped styles

`static styles` holds the component's CSS, written with the `css` tagged template. Lit parses it once per class, not once per instance, and scopes it to the shadow tree.

```
// greeting-message.js
import { LitElement, html, css } from 'lit';

class GreetingMessage extends LitElement {
  static properties = {
    name: {},
  };

  static styles = css`
    :host {
      display: block;
      font-family: system-ui, sans-serif;
    }
```

```
    p {
      margin: 0;
      padding: 8px 12px;
      background: #eef4ff;
      border-radius: 6px;
    }
  `;

  constructor() {
    super();
    this.name = 'there';
  }

  render() {
    return html`<p>Hello, ${this.name}.</p>`;
  }
}

customElements.define('greeting-message', GreetingMessage);
```

The `p` selector styles only the paragraph in this component's shadow tree, the way Chapter 4's shadow `<style>` did: no other `<p>` on the page is affected, and this one needs no class name to protect it. The `:host` rule styles the `<greeting-message>` element itself; `display: block` is there because a custom element defaults to `display: inline`.

Parsing the CSS once per class matters at scale. A page with two hundred `<greeting-message>` elements has one parsed stylesheet shared between them, adopted into each shadow root, not two hundred copies. You get that for free by putting the CSS in `static styles` rather than a `<style>` tag inside the template.

> **Tip.** Keep component CSS in `static styles`, not in a `<style>` element inside `render()`. A `<style>` in the template is re-parsed for every instance and re-processed on every render; `static styles` is parsed once for the whole class.

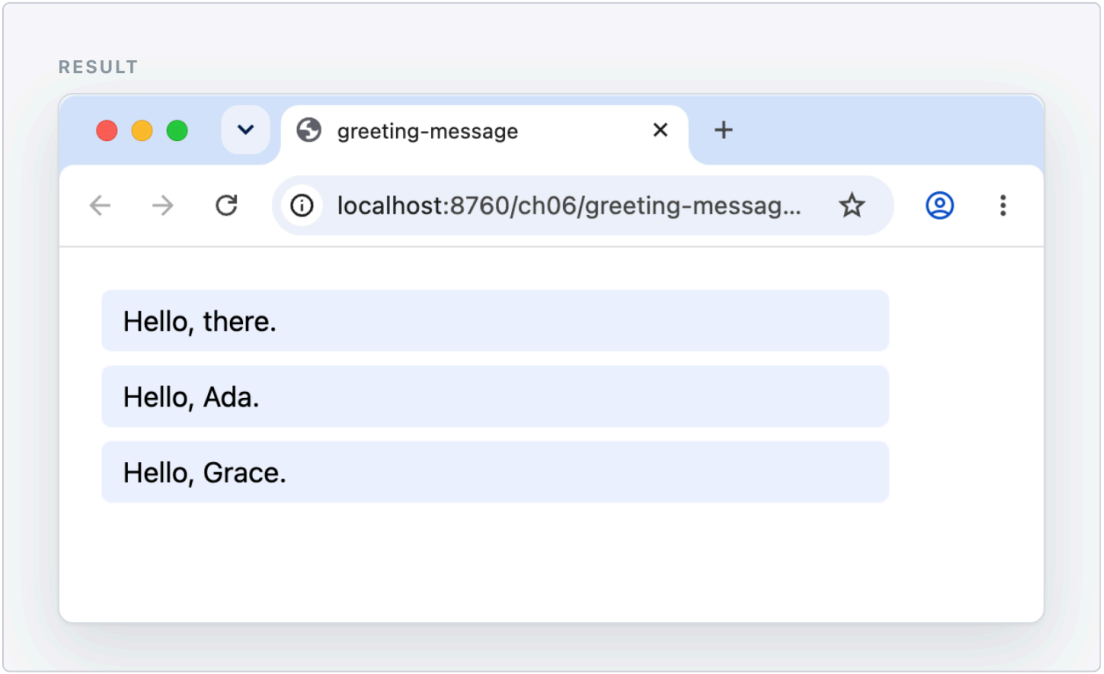

Figure 6.2: `<greeting-message>` with a default, an attribute-set value, and a script-set value, styled by `static styles`. Any change to `name` re-renders the bound text.

6.5 What happens when it runs

Put `<greeting-message name="Ada"></greeting-message>` on a page with the module loaded, and here is the sequence, start to finish.

1. **Define.** `customElements.define('greeting-message', GreetingMessage)` registers the class. If the tag was already in the page, the browser upgrades it now; if not, it upgrades as the parser reaches it.
2. **Construct.** The browser creates the instance and runs the constructor. `super()` runs `LitElement`'s constructor, which sets up the update machinery. Your line sets `this.name = 'there'`, then the upgrade applies the `name="Ada"` attribute, so `this.name` is `"Ada"`.
3. **Connect.** `connectedCallback` fires. Lit attaches the shadow root and requests the first update. The update itself is scheduled asynchronously, so the shadow root is empty for now.
4. **First render.** On the next microtask, Lit adopts `static styles` into the shadow root, calls `render()`, takes the `TemplateResult`, and builds the DOM: `<p>Hello, Ada.</p>`. Because updates are batched, any properties set before this point are folded into this one render.
5. **React.** Later, something runs `element.name = 'Grace'`. The generated setter sees the value changed and schedules an update. On the next microtask Lit calls `render()` again, compares the result to the previ-

ous one, and rewrites just the text node between `Hello,` and `.`. The `<p>` element is never recreated.

Step 5 repeats for the life of the element, once per batch of changes. You wrote a class and a `render()` method; the cycle is the base class's job.

6.6 The case study: the first Cinder component

Cinder's first component is `cn-button`. At this stage it does almost nothing: it wraps a native `<button>` and projects a label into it, but it establishes the pattern every later component follows.

```
// cn-button.js
import { LitElement, html, css } from 'lit';

export class CnButton extends LitElement {
  static styles = css`
    :host {
      display: inline-block;
    }
    button {
      font: inherit;
      padding: 6px 14px;
      border: 1px solid #3b5bdb;
      border-radius: 6px;
      background: #3b5bdb;
      color: #fff;
      cursor: pointer;
    }
  `;

  render() {
    return html`<button><slot></slot></button>`;
  }
}

customElements.define('cn-button', CnButton);
```

```
<cn-button>Save changes</cn-button>
```

The `<slot>` is the same mechanism as Chapter 5: the host's text, `Save changes`, is projected into the shadow `<button>`. The class is `exported` because from Chapter 7 on it lives in a package that other modules import. The `cn-` prefix is Cinder's namespace, kept on every component in the library.

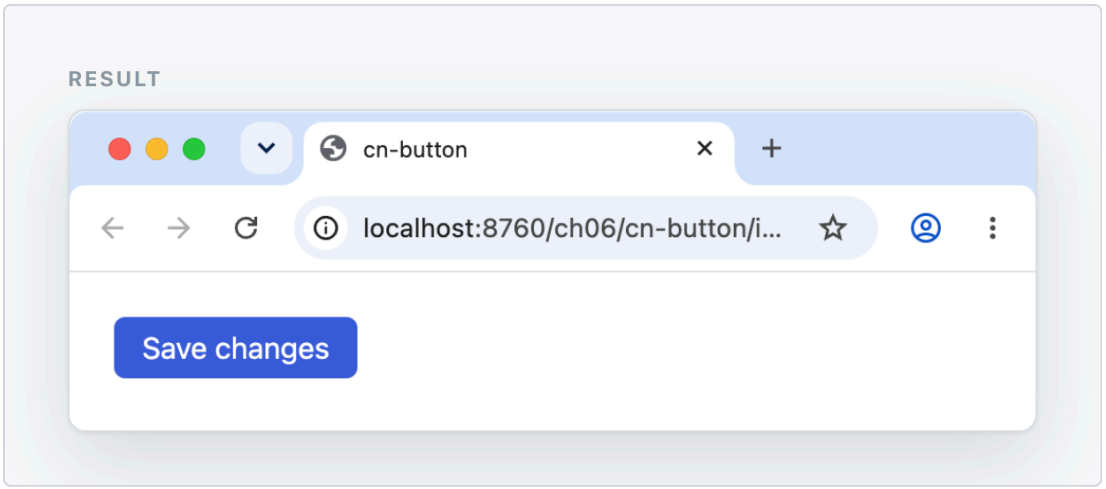

Figure 6.3: `cn-button`, the first Cinder component. It wraps a native `<button>` and slots in the label.

What it's missing (a `disabled` state that really disables the inner button, variants, sizes, an icon slot, focus-ring styling, forwarding clicks) is the material of the chapters ahead. Chapter 7 moves this file into `packages/cinder/` with a real build and test setup; Chapter 24 designs its full API.

Summary

- A Lit component extends `LitElement` and is registered with `customElements.define`. The base class manages the shadow root and the update cycle.
- `render()` returns a `TemplateResult` from the `html` tag: a description of the output, not a string and not a DOM operation. Lit builds the DOM once and patches only what changes after that.
- `static properties = { name: {} }` makes `name` a reactive property: Lit generates the accessor, the attribute wiring, change detection, and a batched re-render. This is the whole of Chapter 3's hand-rolled machinery, per property, in one line.
- `static styles = css`...`` is parsed once per class, scoped to the shadow tree, and shared across all instances.
- The run sequence is define → construct → connect → first render (async) → reactive updates (async, batched). You never call `render()`.
- Cinder starts here with `cn-button`: a `LitElement` wrapping a native `<button>` with a slotted label.

Try it

1. Delete the constructor from `<greeting-message>` and reload. What does the paragraph say, and why?

2. Add `console.count('render')` to `render()`, then set `el.name` twice in one console line. How many renders?

Chapter 7 turns this into a real project (npm, a dev server, the build, and the tests) and moves `cn-button` into the Cinder package.