



CUSTOM ELEMENTS · ACCESSIBILITY · TESTING

Web Components, Hands On

Ten Accessible, Tested
Components Built with **Lit**

Sandeep Kumar Patel

Web Components Handson

Ten Accessible, Tested Components Built with Lit

Sandeep Kumar Patel

Web Components Handson *Ten Accessible, Tested Components Built with Lit*

Copyright © 2026 Sandeep Kumar Patel. All rights reserved.

First edition, 2026.

No part of this book may be reproduced or transmitted in any form or by any means without the prior written permission of the author, except for brief quotations in a review, and except for the source code accompanying the book, which is provided under the terms stated below.

The code. The code listings printed in this book, and the companion code, are released under the MIT License. You may use them in your own projects, commercial or otherwise, without attribution. The companion code is the download described in the Preface, under “Using Code Examples.”

Trademarks. Lit is a trademark of Google LLC. Google Chrome is a trademark of Google LLC. Other product and company names mentioned in this book are the trademarks of their respective owners, used here for identification only.

No warranty. This book and its code are provided “as is,” without warranty of any kind. Web platform features, browser behavior and library APIs change. The code was run and tested against the versions named in the Preface, in Google Chrome, and the reader is responsible for verifying anything load-bearing against the versions and browsers they ship. The author is not liable for any loss or damage arising from the use of this material.

Contents

Preface	5
Who This Book Is For	5
What You Need	5
How This Book Is Organized	5
How We Build Each Component	7
Conventions Used in This Book	7
Using Code Examples	8
Getting Started: Tokens and the Package Template	9
What We Are Building, and Why	9
Technical Requirements	11
Concepts You Need Before Writing Code	11
Step 1: The Token Files	12
Step 2: Themes	13
Step 3: The Component Template	15
Step 4: Tooling	17
Step 5: The First Test	18
Step 6: Enforcing the Token Rules	19
Seeing the Component Run	19
Styling API	22
Accessibility	22
Pitfalls	22
Summary	23
Exercises	23
About this sample	24
 Part I — Compound Components	
1. Tabs	26
What We Are Building, and Why	26
Technical Requirements	27
The Pattern You Are Implementing	28
Concepts You Need Before Writing Code	28
Designing the Markup	33
Step 1: The Tab	33
Step 2: The Panel	35
Step 3: The Parent and Its Slots	36
Step 4: One Function That Makes Everything Agree	37
Step 5: Roving Tabindex	39
Step 6: The Keyboard	39
Step 7: A Cancelable Event	40
Step 8: The Sliding Indicator	42
Seeing the Component Run	43
Styling API	48
Accessibility	49
Testing	50
Pitfalls	51
Summary	52

Exercises	52
Part II — Forms and Input	
4. Combobox	55
What We Are Building, and Why	55
Technical Requirements	57
Concepts You Need Before Writing Code	57
Step 1: The Structure	59
Step 2: Loading Options with a Task	60
Step 3: Keyboard and Selection	62
Step 4: Taking Part in a Form	64
Seeing the Component Run	64
Styling API	67
Accessibility	68
Testing	69
Pitfalls	70
Summary	71
Exercises	71

Preface

This book teaches web components by building ten of them: components that are hard enough to expose the real problems, such as focus management, form participation, keyboard navigation, virtualization and pointer-driven geometry. Each one is built with Lit, a small library for writing custom elements, and each one ships as a standalone package with its own tests and demo page.

Who This Book Is For

This book is for front-end developers who already write JavaScript or TypeScript and want to build components that work in any framework, or in none. You do not need to know Lit in advance, but you should be comfortable with modern JavaScript (classes, modules, async functions), HTML and CSS, and a command line.

It is not an introduction to web components. If custom elements, shadow DOM and slots are new to you, read an introductory text first and return here when you want to build something substantial.

What You Need

- Node 20 or newer, and npm.
- A current version of Google Chrome. This book focuses on Chrome: every test and every screenshot was run in Chromium or Chrome on macOS. The code uses standard platform features, but it has not been tested in other browsers. The components are also browser only: they need a real DOM, and they have not been server-rendered (see Appendix C).
- A code editor with TypeScript support.

The code uses Lit 3.x and TypeScript, and was tested with Lit 3.3.3, `@lit/task` 1.0.3, `@lit/context` 1.1.6 and `@lit-labs/virtualizer` 2.1.1. Platform features that are new or unevenly supported, such as the Popover API or CSS anchor positioning, are called out where they appear. Where the book does not provide a fallback, it says so.

How This Book Is Organized

Getting Started sets up the project template and the design tokens that every later chapter uses. Chapters 1 through 10 each build one component. You can read the chapters in any order, because no component depends on another. If you want a path through them:

If you want to learn about	Read chapters
The essentials of compound components and overlays	Getting Started, 1, 2, 8
Forms and input	Getting Started, 3, 4, 5, 9
Data and hierarchy	Getting Started, 6, 7
Pointer and canvas interaction	Getting Started, 2, 10

The components are:

Chapter	Component	What it teaches
1	Tabs	Compound components, slots, roving tabindex
2	Dialog, popover and tooltip	Layers, focus management, positioning
3	Form-associated inputs	<code>ElementInternals</code> , validation, native form behavior
4	Combobox	The ARIA combobox pattern and async data
5	Date range picker	Calendar math, grid keyboard navigation, locale
6	Data table	Virtualization, sorting, column state, performance
7	Tree view	Recursive rendering, lazy loading, selection models
8	Carousel	Scroll snapping, gestures, accessible autoplay
9	Stepper	Multi-step state, validation gates
10	Image cropper	Canvas, pointer capture, transforms

Six appendices follow the chapters, then a glossary and an index:

Appendix	Subject
A	Token naming rules, and how the token package is published
B	Packaging a component for npm, with a manifest that describes it
C	Using the components in frameworks, and what to check
D	An accessibility testing checklist, and what the tests in this book do and do not establish
E	Solutions to all 55 exercises, each with a test, and what writing them found
F	Further reading: the specifications, pattern guides and documentation the book relied on

How We Build Each Component

Every chapter follows the same six steps, in the same order:

1. **Write the markup you want.** Start from how an author will use the component, before any class exists. The markup is the real API.
2. **Name the state.** Decide the single source of truth, the public properties and the events. Everything else is derived from them.
3. **Build the leaf elements.** Make the smallest pieces first, and make each one render correctly on its own.
4. **Add the coordinator.** Write the parent that finds its children, links them and reacts to input.
5. **Style with tokens.** Components read semantic tokens through their own component tokens, never primitive values.
6. **Test like a user.** Press real keys, run an automated accessibility audit, and cover the edge cases.

Each chapter opens with a short explanation of what the component is and why it is built the way it is, plus an architecture diagram. Two ground rules apply throughout:

- **Every component is standalone.** It lives in its own package and does not import another chapter's component. The only thing components share is the token stylesheet.
- **Every listing was run.** The code printed in this book is copied from the real package files by a script, and the tests pass against those files. If you run the code and get a different result, the book is wrong.

Conventions Used in This Book

The following typographical conventions are used in this book:

- *Italic*: Indicates new terms, file names and emphasis.
- `Constant width`: Used for code, element and attribute names, and CSS properties and tokens.

Tip: This element signifies a tip or suggestion.

Note: This element signifies a general note.

Warning: This element indicates a warning or caution.

Code listings are titled and numbered by chapter, for example *Example 1-3*. Figures are numbered the same way, for example *Figure 1-1*.

Using Code Examples

The code for every chapter is in the companion download, `web-components-handson-code.zip`: one package per component, plus the token package they share. Download it from <https://drive.google.com/file/d/1Ix-JCwVAoB2eOWiRjAH2JgfMZKnYE5dFl/view?usp=sharing>. Each package has its own README, an `npm run dev` demo page and an `npm test` suite. To run one:

```
unzip web-components-handson-code.zip
cd web-components-handson-code/packages/tabs
npm install
npm run dev
```

The tests run in Chromium through Playwright, which downloads the browser separately. Run `npm install playwright chromium` once, and then `npm test`.

The code is released under the MIT License, which the download includes as `LICENSE`. The text of the book is not.

Getting Started: Tokens and the Package Template

Every component in this book starts from the same three things: a project template, a set of design tokens and a handful of rules about how the two fit together. This chapter builds them once, so that each later chapter can begin with the component and skip the setup.

None of it is glamorous. A token file is a list of names and values, and a project template is a few configuration files. But the decisions made here, how values are named, who may read them and how a theme changes them, decide whether ten components look like one family or ten strangers. They also decide whether restyling the whole library takes an afternoon or a month.

In this chapter, you will learn:

- how CSS custom properties pass through a shadow root, and why that makes them the right tool for theming web components
- how to split design tokens into a primitive layer, a semantic layer and a component layer, and what each one is for
- how to write light, dark and high-contrast themes by remapping only the semantic layer
- how to set up a standalone Lit package with TypeScript, Vite and Web Test Runner
- how to write a first component, `ds-button`, that follows the token rules
- how to enforce the token rules with a script, so they cannot drift

What We Are Building, and Why

We are building two things: a **token package** and a **package template**.

The token package, `packages/tokens`, is a few plain CSS files that define every color, space, radius, type size, shadow and duration the components use. The package template, `packages/_template`, is a working Lit package with one small component. Every later chapter copies the template and replaces the component.

Figure GS-1 shows the idea behind the tokens. A raw value lives in a **primitive** token. A **semantic** token gives it a meaning. A **component** token lets one instance override that meaning. The component's CSS reads only the last two.

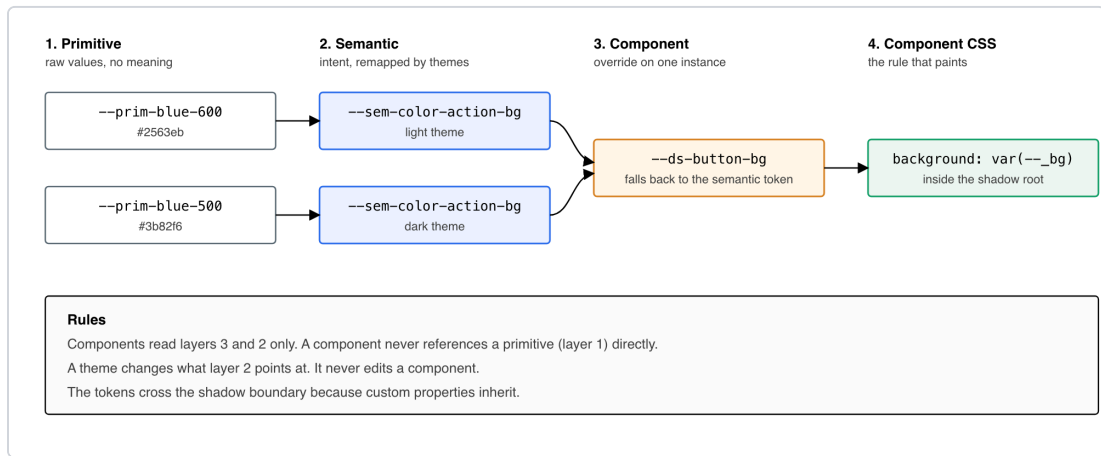


Figure GS-1. How a value travels from a primitive token to the component's CSS

Figure GS-2 shows how the packages relate. Components are independent of one another. The only thing they share is the token stylesheet, and they share it through CSS custom properties, not through JavaScript imports.

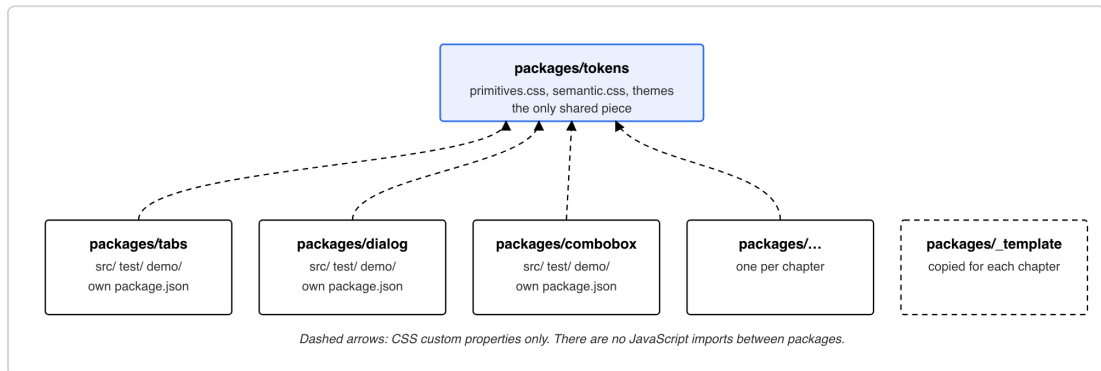


Figure GS-2. The packages in the repository

Why go to this trouble? Because the alternative is hard-coded values. A component with `background: #2563eb` in its CSS looks fine until the brand color changes, or until someone wants a dark theme, or until a customer needs a high-contrast mode. Then every component has to be edited, and every edit is a chance to miss one. A component that reads a token changes when the token changes, and nothing else has to.

Technical Requirements

You need Node 20 or newer, npm, and a current version of Google Chrome. The code for this chapter lives in `packages/tokens` and `packages/_template`.

```
cd packages/_template
npm install
npm run dev    # a demo page with a theme switcher
npm test       # four tests, including an accessibility check
```

Note: In the book’s repository, dependencies are installed once at the root. Each package’s `package.json` still lists everything it needs, so a package copied out installs and runs on its own.

Concepts You Need Before Writing Code

Two ideas carry this chapter: how a custom property crosses a shadow boundary, and how design tokens are layered. Both are short, and everything else in the book rests on them.

Custom Properties and the Shadow Boundary

A **CSS custom property** is a variable you define yourself. Its name starts with two hyphens, and you read it with `var()`:

```
:root { --brand: #2563eb; }
button { background: var(--brand); }
```

Two properties of custom properties matter for web components.

First, they **inherit**. A custom property set on an element is visible to all of that element’s descendants, and that includes descendants inside a shadow root. A shadow root blocks ordinary CSS selectors, but it does not block inheritance. A value set on `:root`, or on a single `<ds-button>`, flows into the component’s shadow tree and can be read by its internal CSS.

Second, `var()` accepts a **fallback**: `var(--x, red)` uses `red` when `--x` is not defined. Fallbacks can be nested, which is how this book lets a component token fall back to a semantic token.

Together, these two properties give a component a styling interface that outsiders can use without breaking encapsulation. The component documents which custom properties it reads, and the page sets them. There is no need to reach into the shadow tree.

Design Tokens and Their Layers

A **design token** is a named design decision: the primary action color, the standard gap between controls, how long a transition takes. Tokens are stored as data and used by name, so a decision is made in one place.

This book uses three layers.

- **Primitive tokens** are raw values with no meaning attached, for example `--prim-blue-600: #2563eb`. They are a palette. You would not guess from the name what the color is for.
- **Semantic tokens** say what a value is for, for example `--sem-color-action-bg`. A semantic token points at a primitive. It answers “what color is an action background?”
- **Component tokens** are the hooks one component exposes, for example `--ds-button-bg`. They default to a semantic token and let a page override a single instance.

The separation pays off in two places. A theme only has to change what the semantic layer points at. And a component that reads `--sem-color-action-bg` automatically follows the theme, because it never learned which blue was involved.

Warning: The rule that components never reference primitive tokens is easy to break by accident, because primitives are right there and `--prim-blue-600` looks like exactly what you want. A single violation hard-wires that component to one theme. Later in the chapter a script enforces the rule.

Step 1: The Token Files

The primitive layer holds the palette, spacing scale, radii, type scale, shadows and motion values. Excerpts follow.

Example GS-1. Excerpt from `primitives.css`

```
--prim-blue-100: #dbeafe;
--prim-blue-300: #93c5fd;
--prim-blue-500: #3b82f6;
--prim-blue-600: #2563eb;
--prim-blue-700: #1d4ed8;

/* Space (4px base) */
--prim-space-0: 0;
--prim-space-1: 0.25rem;
--prim-space-2: 0.5rem;
--prim-space-3: 0.75rem;
--prim-space-4: 1rem;
--prim-space-6: 1.5rem;
--prim-space-8: 2rem;
```

The names are numbers or ordinal steps, not purposes. `--prim-space-4` is not “the standard padding”; it is the fourth step of the scale. The scale is multiples of four pixels, expressed in `rem` so that it grows with the user’s font size.

The semantic layer gives those values meaning:

Example GS-2. Excerpt from semantic.css

```
/* Surfaces */
--sem-color-surface: var(--prim-gray-0);
--sem-color-surface-raised: var(--prim-gray-0);
--sem-color-surface-sunken: var(--prim-gray-100);
--sem-color-overlay-backdrop: rgb(15 23 42 / 0.5);

/* Actions */
--sem-color-action-bg: var(--prim-blue-600);
--sem-color-action-bg-hover: var(--prim-blue-700);
--sem-color-action-subtle-bg: var(--prim-blue-100);
--sem-color-action-subtle-text: var(--prim-blue-700);
```

Each semantic token names a role (surface, text-muted, border, action-bg) and points at a primitive with `var()`. The names say how the value is used, not what it looks like. Note that the file also holds shape (`--sem-radius-control`), spacing (`--sem-space-stack-md`), type, elevation and motion tokens. Color is only the most visible kind.

Motion gets special treatment at the end of the file:

Example GS-3. Reduced motion handled in the semantic layer

```
@media (prefers-reduced-motion: reduce) {
  :root {
    --sem-motion-duration-fast: 0ms;
    --sem-motion-duration: 0ms;
  }
}
```

When the user's system asks for reduced motion, the durations become zero. Every component that reads `--sem-motion-duration` stops animating without containing a media query of its own. One rule serves the whole library.

Step 2: Themes

A theme is a stylesheet that **remaps the semantic layer** and nothing else.

Example GS-4. The dark theme

```
/* Dark theme: remaps the semantic layer only. Applies by OS preference or [data-theme="dark"]. */
@media (prefers-color-scheme: dark) {
  :root:not([data-theme="light"]) {
    --sem-color-surface: var(--prim-gray-900);
    --sem-color-surface-raised: var(--prim-gray-800);
    --sem-color-surface-sunken: var(--prim-gray-1000);
    --sem-color-text: var(--prim-gray-50);
```

```

--sem-color-text-muted: var(--prim-gray-300);
--sem-color-text-danger: var(--prim-red-100);
--sem-color-border: var(--prim-gray-700);
--sem-color-border-strong: var(--prim-gray-500);
--sem-color-focus-ring: var(--prim-blue-300);
--sem-color-action-bg: var(--prim-blue-600);
--sem-color-action-bg-hover: var(--prim-blue-700);
--sem-color-action-subtle-bg: var(--prim-gray-800);
--sem-color-action-subtle-text: var(--prim-blue-300);
}
}
:root[data-theme="dark"] {
--sem-color-surface: var(--prim-gray-900);
--sem-color-surface-raised: var(--prim-gray-800);
--sem-color-surface-sunken: var(--prim-gray-1000);
--sem-color-text: var(--prim-gray-50);
--sem-color-text-muted: var(--prim-gray-300);
--sem-color-text-danger: var(--prim-red-100);
--sem-color-border: var(--prim-gray-700);
--sem-color-border-strong: var(--prim-gray-500);
--sem-color-focus-ring: var(--prim-blue-300);
--sem-color-action-bg: var(--prim-blue-600);
--sem-color-action-bg-hover: var(--prim-blue-700);
--sem-color-action-subtle-bg: var(--prim-gray-800);
--sem-color-action-subtle-text: var(--prim-blue-300);
}

```

The dark theme has two entry points. The `@media (prefers-color-scheme: dark)` block applies it automatically when the operating system is in dark mode, unless the page opted out with `data-theme="light"`. The `:root[data-theme="dark"]` block applies it when the page asks for it, which is what a theme switcher button does. Supporting both means the theme follows the user's system by default and can still be overridden.

The high-contrast theme works the same way, keyed on `prefers-contrast: more` and `data-theme="contrast"`. It darkens text, borders and the action color, makes borders and the focus ring thicker, and turns the hover color nearly black. Because it only remaps semantic tokens, every component picks it up for free.

Step 3: The Component Template

Now the first component. `ds-button` is deliberately small, because its job in this chapter is to show the token pattern, not to be interesting.

Example GS-5. `ds-button.ts`

```
import { LitElement, html, css } from 'lit';
import { customElement, property } from 'lit/decorators.js';

/**
 * Reference component for the package template.
 * Token rule: CSS below uses component tokens (--ds-button-*) only. Those map to
 * SEMANTIC tokens in one block at the top; primitives are never referenced here.
 *
 * @slot - Button label.
 * @csspart button - The native button element.
 * @cssprop --ds-button-bg - Background color.
 * @cssprop --ds-button-fg - Text color.
 * @fires ds-press - Fired when the button is activated.
 */
@customElement('ds-button')
export class DsButton extends LitElement {
  @property({ reflect: true }) variant: 'primary' | 'subtle' = 'primary';
  @property({ type: Boolean, reflect: true }) disabled = false;

  static styles = css`
    :host {
      display: inline-block;
      /* component tokens -> semantic tokens */
      --_bg: var(--ds-button-bg, var(--sem-color-action-bg));
      --_bg-hover: var(--ds-button-bg-hover, var(--sem-color-action-bg-hover));
      --_fg: var(--ds-button-fg, var(--sem-color-text-on-action));
    }
    :host([variant='subtle']) {
      --_bg: var(--ds-button-bg, var(--sem-color-action-subtle-bg));
      --_bg-hover: var(--ds-button-bg-hover, var(--sem-color-surface-sunken));
      --_fg: var(--ds-button-fg, var(--sem-color-action-subtle-text));
    }
    button {
      font: inherit;
      font-family: var(--sem-font-body);
      font-weight: var(--sem-font-weight-control);
      padding: var(--sem-control-padding-block) var(--sem-control-padding-inline);
      border: var(--sem-border-width) solid transparent;
      border-radius: var(--sem-radius-control);
      background: var(--_bg);
      color: var(--_fg);
      cursor: pointer;
      transition: background var(--sem-motion-duration-fast) var(--sem-motion-easing);
    }
    button:hover:not(:disabled) { background: var(--_bg-hover); }
  `;
}
```

```

button:focus-visible {
  outline: var(--sem-focus-ring-width) solid var(--sem-color-focus-ring);
  outline-offset: var(--sem-focus-ring-offset);
}
button:disabled { opacity: 0.5; cursor: not-allowed; }
`;

render() {
  return html`<button part="button" ?disabled=${this.disabled} @click=${this.#press}>
    <slot></slot>
  </button>`;
}

#press() {
  this.dispatchEvent(new CustomEvent('ds-press', { bubbles: true, composed: true }));
}

}

declare global {
  interface HTMLElementTagNameMap { 'ds-button': DsButton }
}

```

How the Code Works

Look at the `:host` rule in `static` styles. It defines **private** custom properties, `--_bg`, `--_bg-hover` and `--_fg`. Each one reads a component token (`--ds-button-bg`) and falls back to a semantic token (`--sem-color-action-bg`). The rest of the CSS uses only the private names.

This is the **component token pattern**, and it is worth the extra line for three reasons:

1. **A single place** states every token the component reads, so the styling API is easy to document.
2. **A variant is a one-line override.** The `:host([variant='subtle'])` rule swaps the three private properties, and every other rule follows.
3. **The private name keeps outsiders out.** A page sets `--ds-button-bg`; the underscore-prefixed names are an implementation detail it should not touch.

Notice also what is absent: no color, size or duration is written as a literal anywhere in the CSS. Even the disabled state's `opacity: 0.5` is a plain value, not a token, because it is not a brand decision.

Note: The component has no fallback value of its own for the semantic tokens. If the token stylesheet is not loaded, the properties are *guaranteed-invalid* and the button renders without a background. That is a deliberate trade: hard-coded fallbacks would break the “no literal values” rule and hide a missing stylesheet.

Step 4: Tooling

Each package carries the same small set of configuration files.

Example GS-6. tsconfig.json

```
{
  "compilerOptions": {
    "target": "ES2022",
    "module": "ES2022",
    "moduleResolution": "bundler",
    "lib": ["ES2022", "DOM", "DOM.Iterable"],
    "strict": true,
    "experimentalDecorators": true,
    "useDefineForClassFields": false,
    "skipLibCheck": true,
    "declaration": true,
    "outDir": "dist",
    "rootDir": "."
  },
  "include": ["src", "test", "demo"]
}
```

Two settings are specific to Lit's decorators. `experimentalDecorators` turns on the TypeScript decorator syntax Lit uses for `@property` and `@customElement`, and `useDefineForClassFields: false` makes class fields behave the way those decorators expect. Without the second one, a decorated property can be silently shadowed by a class field and never trigger an update.

The test runner configuration is short:

Example GS-7. web-test-runner.config.mjs

```
import { playwrightLauncher } from '@web/test-runner-playwright';
import { esbuildPlugin } from '@web/dev-server-esbuild';
import { summaryReporter, defaultReporter } from '@web/test-runner';

export default {
  files: 'test/**/*.test.ts',
  nodeResolve: true,
  plugins: [esbuildPlugin({ ts: true, tsconfig: './tsconfig.json' })],
  reporters: [defaultReporter({ reportTestResults: false }), summaryReporter({ flatten: false })],
  browsers: [playwrightLauncher({ product: 'chromium' })],
};
```

Web Test Runner serves the test files, compiles TypeScript with esbuild, and runs them in real Chromium through Playwright. The `reporters` line only changes the output, so that each test is listed by name. Using a real browser matters for web components, because shadow DOM, slots, focus and custom elements are browser features that simulated DOM environments reproduce only partly.

Step 5: The First Test

The component's test file checks behavior, an event and accessibility.

Example GS-8. `ds-button.test.ts`

```
import { fixture, html, expect, oneEvent } from '@open-wc/testing';
import '../src/ds-button.js';
import type { DsButton } from '../src/ds-button.js';

describe('ds-button', () => {
  it('renders its label and reflects variant', async () => {
    const el = await fixture<DsButton>(html`<ds-button variant="subtle">Hi</ds-button>`);
    expect(el.textContent).to.equal('Hi');
    expect(el.getAttribute('variant')).to.equal('subtle');
  });

  it('fires ds-press on click', async () => {
    const el = await fixture<DsButton>(html`<ds-button>Go</ds-button>`);
    setTimeout(() => el.shadowRoot!.querySelector('button')!.click());
    const ev = await oneEvent(el, 'ds-press');
    expect(ev).to.exist;
  });

  it('does not fire when disabled', async () => {
    const el = await fixture<DsButton>(html`<ds-button disabled>No</ds-button>`);
    let fired = false;
    el.addEventListener('ds-press', () => (fired = true));
    el.shadowRoot!.querySelector('button')!.click();
    expect(fired).to.be.false;
  });

  it('is accessible', async () => {
    const el = await fixture<DsButton>(html`<ds-button>Save</ds-button>`);
    await expect(el).to.be.accessible();
  });
});
```

The last test is the interesting one. `expect(el).to.be.accessible()` runs the axe-core engine against the rendered element and fails on violations. It will not catch everything, since automated tools find only a portion of accessibility problems, but it catches the mechanical ones early and costs one line.

Run the suite:

```
npm test
```

Example GS-9. Output of `npm test` in the template package

```
$ npm test

ds-button [Chromium]
  ✓ renders its label and reflects variant
  ✓ fires ds-press on click
  ✓ does not fire when disabled
  ✓ is accessible
🚧 Browser logs:
  Lit is in dev mode. Not recommended for production! See https://lit.dev/msg/dev-mode for
  more information.
Finished running tests in 0.2s, all tests passed! 🎉
```

Step 6: Enforcing the Token Rules

A rule that only lives in a book gets broken. This repository enforces the token rules with a short script that scans every package.

Example GS-10. Rule 1 of scripts/check-tokens.mjs: components never use primitives

```
// Rule 1: components never use primitives.
for (const pkg of readdirSync(PACKAGES)) {
  if (pkg === 'tokens') continue;
  const srcDir = join(PACKAGES, pkg, 'src');
  if (!existsSync(srcDir)) continue;
  for (const file of walk(srcDir).filter((f) => /\.?(ts|css)$/.test(f))) {
    readFileSync(file, 'utf8').split('\n').forEach((line, i) => {
      if (/--prim-/\.test(line)) problems.push(`${relative(ROOT, file)}:${i + 1} uses a
primitive token: ${line.trim()}`);
    });
  }
}
```

Run it with `npm run check:tokens` from the repository root. It exits with status 1 and prints the file and line of any component source that mentions `--prim-`. A second rule in the same script checks that the semantic layer references only primitives, so no semantic token silently depends on a component token. Because the script exits non-zero on a violation, a continuous integration job can use it to block a merge.

Tip: Write the rule check *before* you need it. A new package is easiest to keep clean from its first commit.

Seeing the Component Run

The screenshots in this section are real, made in the same way as those in Chapter 1: a script opens the demo page in Google Chrome with a clean profile, drives the component with real mouse and keyboard input through Playwright, and captures the Chrome window. Nothing here is a mockup. To reproduce them on macOS, run `node scripts/capture-chapter.mjs ch00` from the book's repository.

Figure GS-3 is the template component in its default state, the light theme. Every color in it comes from a semantic token, read through a component token.

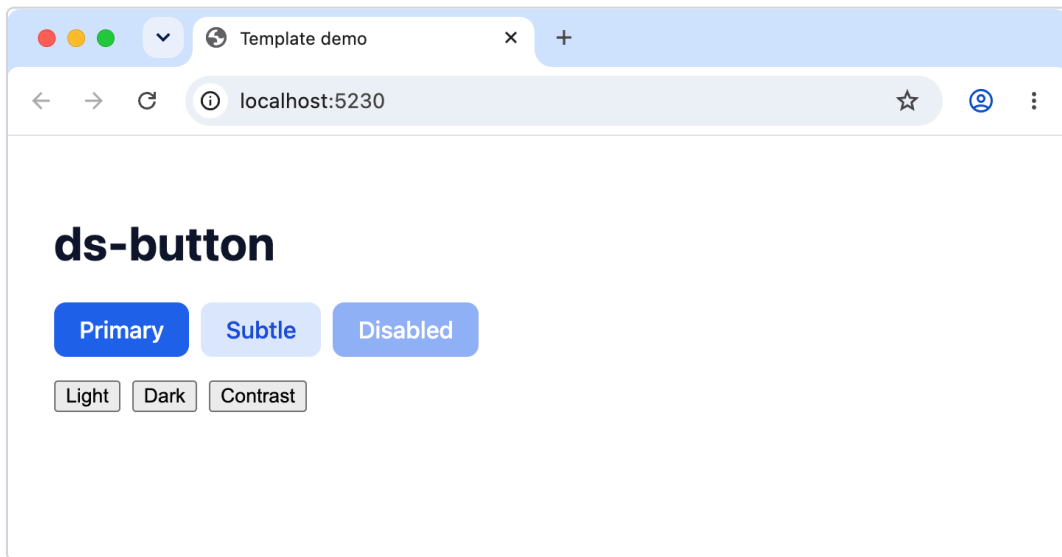


Figure GS-3. The template button in the light theme

Figure GS-4 is the same page after a click on the Dark button. The component code did not change. The page set `data-theme="dark"` on the document element, which remaps the semantic tokens.

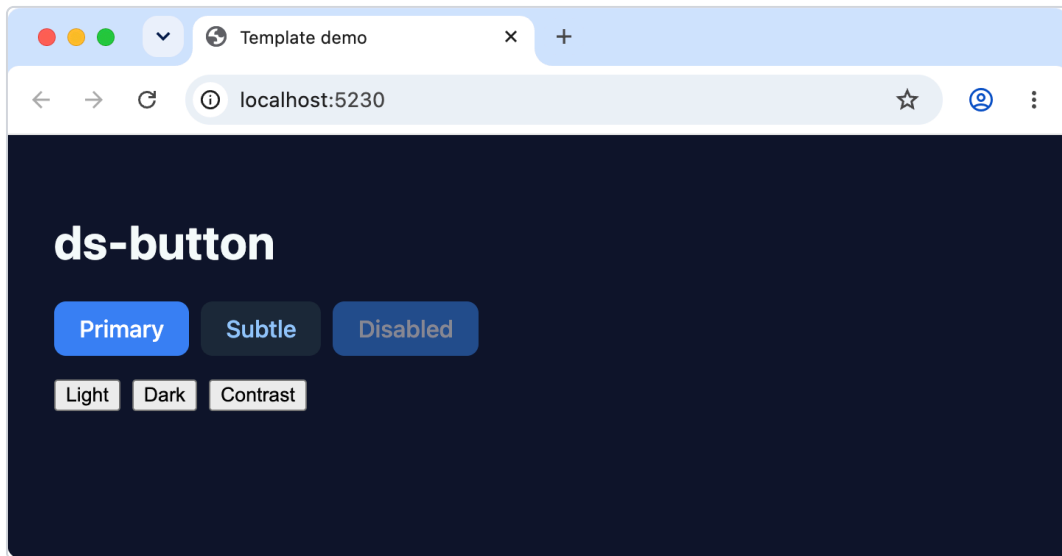


Figure GS-4. The same button in the dark theme

Figure GS-5 applies the high-contrast theme in the same way. The change is subtle on this page, because the button has no visible border to thicken: the action color and the text are darker. A theme is only a set of overrides for semantic tokens, so adding one never touches a component.

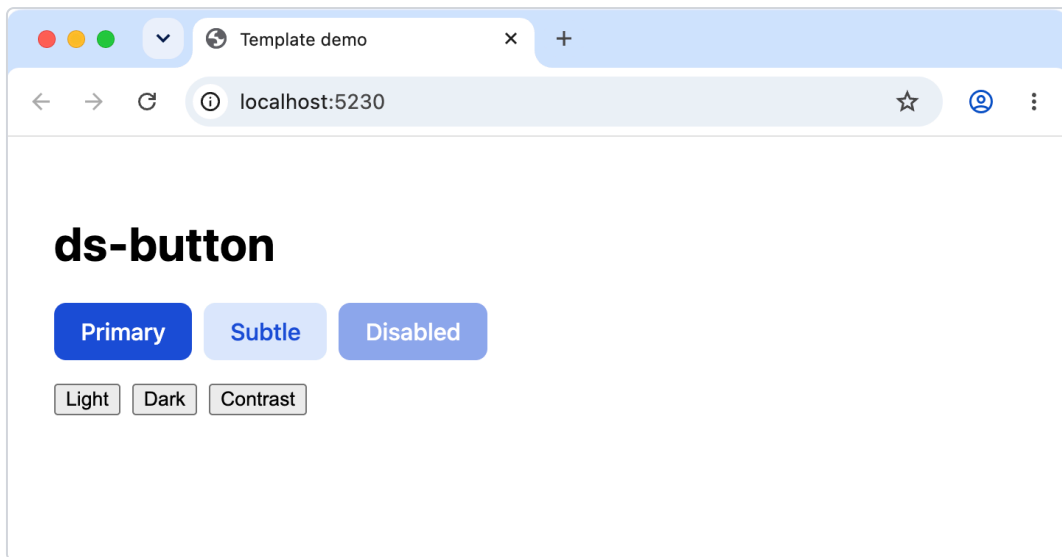


Figure GS-5. The same button in the high-contrast theme

Styling API

This chapter's component establishes the pattern every later chapter documents. Each component lists the same four things:

- the **semantic tokens** it consumes
- the **component tokens** it exposes
- its `::part()` **names**, which expose elements for structural styling
- its **slots**

For `ds-button`:

Kind	Name	Purpose
Component token	<code>--ds-button-bg</code>	Background
Component token	<code>--ds-button-bg-hover</code>	Background on hover
Component token	<code>--ds-button-fg</code>	Text color
Part	<code>button</code>	The native button element
Slot	<code>default</code>	The label
Event	<code>ds-press</code>	Fired on activation

Accessibility

Two accessibility properties come from the token layer rather than from any one component. The **focus ring** is drawn with `--sem-focus-ring-width` and `--sem-color-focus-ring`, and the dark and high-contrast themes remap the color so the ring stays visible on every background. And the **motion** tokens go to zero under `prefers-reduced-motion`, so no component needs to remember to check.

Tokens cannot supply *contrast between* a text color and its background, which depends on the pairs a component uses. The axe test in each package checks it.

Pitfalls

Setting a token on the wrong element. A custom property set on `body` reaches every component below it. One set on a sibling does not. When a token seems to be ignored, check that the element that sets it is an ancestor of the component.

Reading a token that nothing defines. A typo such as `--sem-color-acton-bg` is not an error: the property is undefined, and the declaration that used it is dropped. DevTools flags it in the Styles pane. The token check cannot catch this, so name tokens carefully and test themes in the browser.

Theming by overriding component CSS. If you find yourself writing `ds-button::part(button) { background: ... }` to change a theme, a token is missing. Add the token and let the component read it.

Letting primitives leak. A page using `--prim-blue-600` directly in its own CSS defeats the theme for that rule. Keep primitives out of everything except the semantic layer.

Summary

In this chapter, you built the foundation that every later chapter reuses. Custom properties inherit through shadow roots, which makes them the right interface between a page and a component. The token system has three layers: primitive values, semantic roles and per-component hooks, and only the last two are ever read by components. Themes remap the semantic layer and change nothing else. A standalone package template wraps a component, its tests and its demo, and a script keeps the token rules from drifting.

In the next chapter, you'll use this setup to build your first compound component, a set of tabs that coordinates a tab list and its panels.

Exercises

Each exercise extends this chapter's component. Add a test where the behavior can be tested. Worked solutions, each with a test, are in section E.2 of Appendix E and in `packages/_template/solutions/`.

1. **A new semantic token.** Add `--sem-color-danger-text` to the semantic layer and make the dark theme remap it. Use it in a `variant="danger"` for `ds-button`, with a test that checks the computed color changes between themes.
2. **A third theme.** Write a `theme-sepia.css` that remaps only semantic tokens, then add it to the demo page's switcher. Confirm that `npm run check:tokens` still passes.
3. **Break the rule on purpose.** Use `--prim-blue-600` in `ds-button` and run the check script. Read its output, then revert the change.
4. **A disabled opacity token.** Replace the literal `opacity: 0.5` with `--sem-opacity-disabled`, defined in the primitive and semantic layers, and use it from the button.

About this sample

This is a free sample of *Web Components, Hands On*: the front matter, the Getting Started chapter, Chapter 1 (tabs, the pilot that introduces every concept the later chapters reuse), and Chapter 4 (the combobox, with asynchronous data). The full book builds ten components, each as a standalone, tested Lit package on a shared token layer, and ends with six appendices, a glossary and an index.



PART I

Compound Components



1 Tabs

Chapter 1. Tabs

Tabs look like the easiest component in any design system. A row of labels, a block of content, a click that swaps one for the other. A first version takes ten minutes.

The trouble starts when someone presses a key. Keyboard users expect arrow keys to move between tabs, Home and End to jump to the ends, and Tab to leave the tab list rather than walk through every label. Screen reader users expect the browser to announce “tab, 2 of 4, selected”. None of that comes for free, and all of it has to survive when the tabs and panels are separate elements that a parent has to coordinate.

That coordination is the real subject of this chapter. A tabs component is a parent that manages children it did not create: it finds them, links them together, keeps them in sync with one piece of state, and reacts to events from them. Nearly every compound component in this book (tree, stepper, carousel) follows the same shape, so Tabs is the best place to learn it.

In this chapter, you will learn:

- the platform concepts underneath it: custom elements, shadow DOM, slots, Lit’s update cycle, events and the accessibility tree
- how to build a compound component from three cooperating custom elements
- how named slots and `@queryAssignedElements` let a parent find its children
- how to implement roving tabindex so the tab list is a single stop in the page’s tab order
- why ARIA relationships (`aria-controls`, `aria-labelledby`) are written as attributes by the parent
- how automatic and manual activation differ, and how to support both
- how to fire a cancelable event for a change the user triggers
- how to animate a selection indicator by measuring layout
- how to test keyboard behavior with real key presses and an automated accessibility check

What We Are Building, and Why

A tabs component lets a user switch between several views of related content while only one is visible at a time. Product pages use it for “Overview / Specs / Support”, settings screens use it for categories, and dashboards use it for time ranges.

We build it as a web component, rather than copying a framework’s version, for three reasons. A web component is plain HTML to its users, so the same `<ds-tabs>` works in a React app, a Vue app, a server-rendered page or a static file. It carries its own encapsulated styles, so a stray global rule cannot break it. And it forces you to deal with the web platform directly (slots, shadow DOM, focus, events), which is what the rest of this book builds on.

The component is three custom elements that cooperate:

Element	Role
<code><ds-tabs></code>	The coordinator. Owns the selected state, the keyboard handling and the indicator.
<code><ds-tab></code>	One label in the tab list.
<code><ds-tab-panel></code>	The content shown when its tab is selected.

Figure 1-1 shows how they are arranged. Read it from left to right: the author writes tabs and panels as siblings (box 1), the parent's shadow root provides a slot for each group (box 2), and a small amount of state and logic inside the parent keeps everything in agreement (box 3). The strip underneath is the path every change takes, whether it started from a click, a key press, a tab added later, or a line of code.

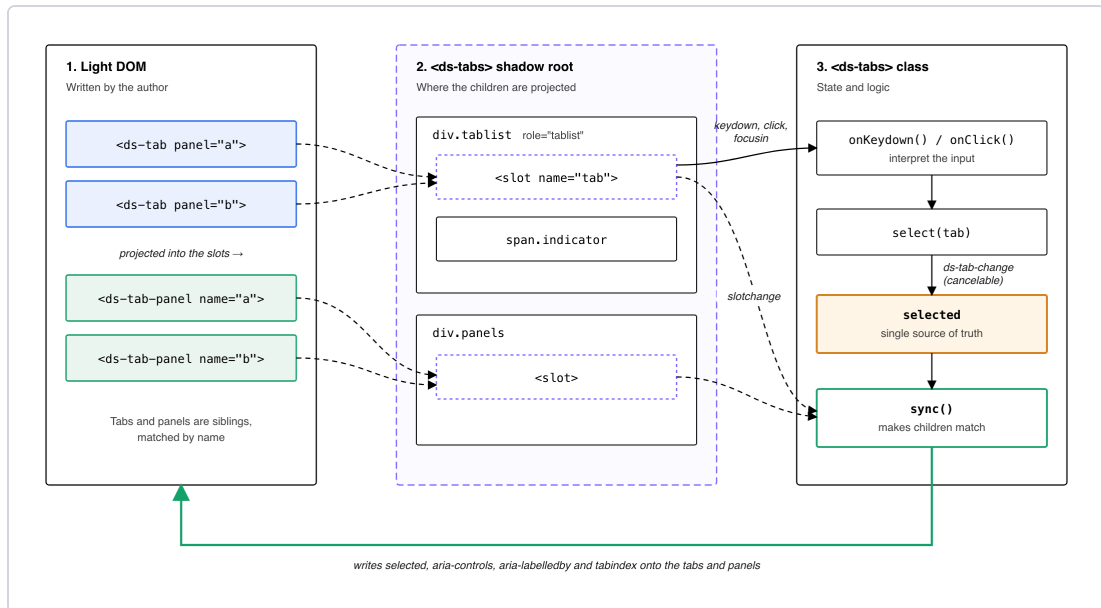


Figure 1-1. How the three tab elements fit together

Keep this picture in mind. Each build step later in the chapter constructs one part of it.

Technical Requirements

You need Node 20 or newer and a current version of Google Chrome. The code for this chapter lives in `packages/tabs`:

```
cd packages/tabs
npm install
npm run dev      # demo page with all three variants
npm test        # 12 tests, including an accessibility check
```

The component reads the design tokens from `packages/tokens`: primitive values such as `--prim-blue-600`, and the semantic tokens built on them such as `--sem-color-action-bg`. Components only ever read the semantic layer. The demo page loads the token files for you.

The Pattern You Are Implementing

The WAI-ARIA Authoring Practices Guide describes tabs as three roles that work together:

Role	Element	Key attributes
<code>tablist</code>	the container that holds the tabs	<code>aria-orientation</code>
<code>tab</code>	each label	<code>aria-selected</code> , <code>aria-controls</code> , <code>aria-disabled</code>
<code>tabpanel</code>	each block of content	<code>aria-labelledby</code>

The keyboard model is part of the pattern, not an extra:

- **Tab** moves focus into the tab list, landing on the selected tab, and the next press moves focus out to the selected panel and beyond. The tab list is one stop.
- **Arrow keys** (Left and Right for horizontal tabs, Up and Down for vertical) move focus between tabs.
- **Home** and **End** move to the first and last tab.
- With **automatic activation**, moving focus also selects the tab. With **manual activation**, focus moves first, and **Enter** or **Space** selects.

Which activation model is right depends on how expensive a panel is. If showing a panel is instant, automatic is friendlier. If each panel loads data or runs a heavy render, manual lets a keyboard user pass over tabs without triggering that work. Our component supports both through one attribute.

Concepts You Need Before Writing Code

The component in this chapter is short, but it leans on six ideas from the web platform and from Lit. Each one is explained here in the terms the code will use. If you already know an idea well, skim its summary line and move on.

Custom Elements

A custom element is an HTML element that you define yourself. You pick a name that contains a hyphen, so it can never collide with a built-in tag, and register a class for it:

```
class MyThing extends HTMLElement {}
customElements.define('my-thing', MyThing);
```

From then on, every `<my-thing>` in the document is an instance of that class. The browser calls methods on the class at defined moments. The two you will meet in this chapter are `connectedCallback`, which runs when the element is inserted into a document, and `disconnectedCallback`, which runs when it is removed.

An element can appear in the markup before its class is registered. The browser treats it as an unknown element until `customElements.define` runs, then *upgrades* it in place. This is why custom elements work in server-rendered HTML and why markup order rarely matters.

Writing the class by hand against `HTMLElement` means managing attributes, rendering and updates yourself. Lit is a thin base class, `LitElement`, that does that bookkeeping for you.

Light DOM, Shadow DOM and the Flattened Tree

Every element can have two kinds of children.

The **light DOM** is the ordinary DOM: whatever the author writes between the tags. For `<ds-tabs>`, that is the tabs and the panels.

A **shadow tree** is a second, private DOM that a component attaches to its own element. Lit creates one for you automatically. Anything inside it is encapsulated: CSS from the outer page does not leak in, the component's own CSS does not leak out, and `document.querySelector` cannot see inside it.

A component usually wants both. It wants to draw its own structure (a tab list, an indicator) in the shadow tree, while showing content that the author supplied in the light DOM. The bridge between the two is the **slot**.

The browser combines the two trees into a **flattened tree** by replacing each `<slot>` with the light DOM nodes assigned to it. Figure 1-2 shows the three trees side by side for our component.



Figure 1-2. The light DOM and the shadow tree combine into the flattened tree

Two facts about the flattened tree explain a lot of what follows:

- The browser renders the flattened tree, and the accessibility tree is built from it. That is why a `ds-tab` that lives in the light DOM can count as a child of the `role="tablist"` element that lives in the shadow tree.
- DOM APIs mostly see one tree at a time. Table 1-1 shows which tree each call sees.

Table 1-1. Which tree each API sees

API	Sees
<code>host.querySelector('ds-tab')</code>	The light DOM children
<code>host.shadowRoot.querySelector('.tablist')</code>	The shadow tree
<code>slot.assignedElements()</code>	The light DOM nodes assigned to that slot
<code>element.closest('ds-tab')</code>	Ancestors in the tree the element belongs to
<code>event.composedPath()</code>	The path through the flattened tree, including slots

Note: Lit attaches an *open* shadow root, so `host.shadowRoot` is available from outside. Tests in this book rely on that.

You can see these trees in a real browser, too. Figure 1-9, later in the chapter, shows the Chrome DevTools Elements panel for the finished component.

Styling follows the same boundary. A rule written inside the component's shadow tree applies to the shadow tree's own elements, but it does not reach the slotted light DOM children, which are styled by the outer page and by their own shadow trees. Each of our three elements carries its own styles for exactly that reason.

Slots

A `<slot>` is a placeholder in a shadow tree. There are two kinds:

- A **default slot**, written `<slot></slot>`, receives every light DOM child that does not ask for a named slot.
- A **named slot**, written `<slot name="tab"></slot>`, receives the children that carry a matching attribute, `slot="tab"`.

Assignment is automatic and live. If a child is added, removed or given a different `slot` attribute, the browser reassigns it and fires a `slotchange` event on the affected slot. Code that needs to know which elements are in a slot calls `slot.assignedElements()`, and Lit wraps that call in a decorator, `@queryAssignedElements`, which you will use in Step 3.

Two details matter for this component:

- **`slotchange` fires after the browser has assigned nodes**, including for the first render. It is the reliable signal that the children are ready, which `connectedCallback` is not.
- **The `slot` attribute is read from the child**. The parent cannot decide where a child goes without the child's cooperation, which is why `ds-tab` sets its own `slot` when it connects.

Lit's Update Cycle

Lit turns plain class properties into **reactive properties**. When you assign to one, Lit does not render immediately. It schedules an update, and all the changes made during the same tick are batched into one render.

You declare a reactive property with a decorator:

```
@property({ type: Boolean, reflect: true }) selected = false;
```

Here `@property` makes `selected` reactive. `type: Boolean` tells Lit how to convert between the attribute (present or absent) and the property (`true` or `false`). `reflect: true` writes the property's value back to the HTML attribute, so `el.selected = true` produces `<ds-tab selected>`. Reflection is what lets CSS write selectors such as `:host([selected])`.

A scheduled update runs a short sequence of lifecycle methods. Table 1-2 lists the ones this chapter uses.

Table 1-2. The Lit lifecycle methods used in this chapter

Method	When it runs	Used here for
<code>connectedCallback</code>	The element is added to a document	Setting <code>slot</code> and <code>role</code> ; starting the <code>ResizeObserver</code>
<code>willUpdate(changed)</code>	Before every render	Deriving state that depends on properties, such as <code>aria-selected</code>
<code>render()</code>	Each update	Returning the shadow tree's template
<code>updated(changed)</code>	After the DOM has been updated	Measuring layout for the indicator
<code>disconnectedCallback</code>	The element is removed	Stopping the <code>ResizeObserver</code>

The `changed` argument is a map of the properties that changed and their previous values, so `changed.has('selected')` asks “did `selected` change in this update?”

Note: The decorators in this book use TypeScript's `experimentalDecorators` setting, together with `useDefineForClassFields: false`. Both are set in each package's `tsconfig.json`. Lit also supports the newer standard decorators, which need a slightly different setup.

Property assignments made inside `willUpdate` are safe, because they are folded into the update that is already in progress. Assignments made inside `updated` trigger another update, and Lit warns about them in development mode. That is why the indicator is positioned with plain DOM calls and not with reactive state.

Events, Bubbling and Composition

When a user clicks or presses a key, the browser creates an event and sends it along a **path**: from the target element up through its ancestors to the document. Listeners along the path run in order. This upward travel is called **bubbling**, and it enables **event delegation**: one listener on a parent can handle events for every child, including children added later.

Shadow trees add two rules:

- The path follows the flattened tree. An event from a slotted `ds-tab` passes through its `<slot>` and on to the slot's ancestors in the shadow tree, so a listener on `div.tablist` receives it.
- An event only crosses a shadow boundary if it is **composed**. Native events such as `click` and `keydown` are composed. Events you create yourself are not, unless you set `composed: true`.

You create your own events with `CustomEvent`:

```
el.dispatchEvent(new CustomEvent('ds-tab-change', {
  detail: { selected: 'b' },
  bubbles: true,
  composed: true,
  cancelable: true,
}));
```

`detail` carries data. `bubbles` and `composed` let the event travel up and out of any shadow trees. `cancelable` allows a listener to call `preventDefault()`, and `dispatchEvent` then returns `false`, which is how a component can ask “may I proceed?” without owning the answer.

Accessibility Trees, ARIA and Focus

Browsers build a second tree from the page, the **accessibility tree**, and send it to assistive technology such as screen readers. Each node has a **role** (what it is), a **name** (what it is called) and **states and properties** (what it is doing). A native `<button>` has the role `button` and is focusable without any help. A custom element has no role at all until you give it one.

ARIA attributes supply that information. This chapter uses three kinds:

- **Roles**: `tablist`, `tab` and `tabpanel` say what each element is.
- **States**, which change as the user interacts: `aria-selected` and `aria-disabled`.
- **Relationships**, which point at other elements by id: `aria-controls` (“this tab controls that panel”) and `aria-labelledby` (“this panel is named by that tab”).

ARIA changes only what assistive technology is told. It does not add behavior. Setting `role="tab"` does not make anything focusable or respond to arrow keys; the component has to implement both.

Which brings us to **focus**. The browser's **tab order** is the sequence of elements reached by pressing Tab. Two values of the `tabindex` attribute control membership:

- `tabindex="0"` puts an element in the tab order, at its natural position.
- `tabindex="-1"` takes it out of the tab order but still lets code or a click focus it.

A tab list manages this itself, giving one tab 0 and the rest -1. The technique is called roving tabindex, and Step 5 covers it.

Warning: ID references such as `aria-controls` resolve only inside a single tree. An element in a shadow tree cannot point at an element in the light DOM by `id`, or the other way around. Step 4 shows how the component works within this rule.

Designing the Markup

Write the markup you would want to author before writing any code:

```
<ds-tabs>
  <ds-tab panel="overview">Overview</ds-tab>
  <ds-tab panel="specs">Specs</ds-tab>
  <ds-tab-panel name="overview">...</ds-tab-panel>
  <ds-tab-panel name="specs">...</ds-tab-panel>
</ds-tabs>
```

Three design decisions are visible here.

Tabs and panels are siblings, matched by name. The alternative is to pair them by position, so the third tab controls the third panel. Position is fragile: reorder the markup or conditionally render one tab and every pair after it shifts. A name is explicit and survives edits.

The author never writes `slot="tab"`. The parent needs tabs in one place (the tab list) and panels in another, so it uses two slots. But requiring authors to remember a `slot` attribute on every tab is a trap. The tab sets its own `slot` when it connects.

selected lives on the parent. One attribute on `<ds-tabs>` holds the name of the selected panel. Individual tabs and panels get a `selected` property too, but they are outputs the parent writes, not inputs the author sets.

Step 1: The Tab

Start with the leaf elements, because they are the simplest. A tab is a label that knows which panel it belongs to and whether it is selected or disabled:

Example 1-1. `ds-tab.ts`

```
import { LitElement, html, css } from 'lit';
import { customElement, property } from 'lit/decorators.js';

/**
 * A single tab button. Place it inside <ds-tabs>; it is slotted into the tab list.
 *
 * @slot - Tab label.
 * @cssprop --ds-tab-fg - Label color.
 * @cssprop --ds-tab-fg-selected - Label color when selected.
```

```

*/
@customElement('ds-tab')
export class DsTab extends LitElement {
  /** Name of the ds-tab-panel this tab controls. */
  @property({ reflect: true }) panel = '';
  @property({ type: Boolean, reflect: true }) selected = false;
  @property({ type: Boolean, reflect: true }) disabled = false;

  static styles = css`
    :host {
      --_fg: var(--ds-tab-fg, var(--sem-color-text-muted));
      --_fg-selected: var(--ds-tab-fg-selected, var(--sem-color-action-subtle-text));
      display: inline-block;
      flex: none;
      padding: var(--sem-control-padding-block) var(--sem-control-padding-inline);
      font-family: var(--sem-font-body);
      font-size: var(--sem-font-size-body);
      font-weight: var(--sem-font-weight-control);
      color: var(--_fg);
      cursor: pointer;
      user-select: none;
      transition: color var(--sem-motion-duration-fast) var(--sem-motion-easing);
    }
    :host(:hover:not([disabled]):not([selected])) { color: var(--sem-color-text); }
    :host([selected]) { color: var(--_fg-selected); }
    :host([disabled]) { opacity: 0.5; cursor: not-allowed; }
    :host(:focus-visible) {
      outline: var(--sem-focus-ring-width) solid var(--sem-color-focus-ring);
      outline-offset: calc(var(--sem-focus-ring-width) * -1);
      border-radius: var(--sem-radius-control);
    }
  `;

  connectedCallback() {
    super.connectedCallback();
    this.slot = 'tab';
    this.setAttribute('role', 'tab');
  }

  protected willUpdate() {
    this.setAttribute('aria-selected', String(this.selected));
    this.setAttribute('aria-disabled', String(this.disabled));
  }

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

```

```

    }
  }

  declare global {
    interface HTMLElementTagNameMap { 'ds-tab': DsTab }
  }

```

How the Code Works

The three `@property` declarations are the tab's public state. `reflect: true` writes each one back to an attribute, which is what lets the CSS select on `:host([selected])` and lets the parent query the DOM.

`connectedCallback` runs when the element enters the document. It sets `slot` to `'tab'` (so the parent's named slot picks it up) and sets `role="tab"`. `willUpdate` runs before every render and keeps `aria-selected` and `aria-disabled` in step with the properties. Because the ARIA state is derived from properties inside `willUpdate`, there is no way for the two to disagree.

The styles never mention a primitive token. Each color is read through a component token such as `--ds-tab-fg`, which falls back to a semantic token. Someone restyling a single tabs instance sets `--ds-tab-fg`; a theme changes the semantic layer. Neither has to touch the component.

The render method is only a `<slot>`. A tab has no internal structure; its label is whatever the author puts inside it.

Step 2: The Panel

The panel is even smaller. It has a name, a selected flag and a role:

Example 1-2. ds-tab-panel.ts

```

import { LitElement, html, css } from 'lit';
import { customElement, property } from 'lit/decorators.js';

/**
 * Content for one tab. Hidden unless its tab is selected.
 *
 * @slot - Panel content.
 */
@customElement('ds-tab-panel')
export class DsTabPanel extends LitElement {
  /** Matches the `panel` attribute of its ds-tab. */
  @property({ reflect: true }) name = '';
  @property({ type: Boolean, reflect: true }) selected = false;

  static styles = css`
    :host { display: block; padding: var(--ds-tabs-panel-padding, var(--sem-space-stack-md)); }
    :host(:not([selected])) { display: none; }
    :host(:focus-visible) {
      outline: var(--sem-focus-ring-width) solid var(--sem-color-focus-ring);
    }
  `;
}

```

```

        outline-offset: calc(var(--sem-focus-ring-width) * -1);
    }
`;

connectedCallback() {
    super.connectedCallback();
    this.setAttribute('role', 'tabpanel');
}

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

declare global {
    interface HTMLElementTagNameMap { 'ds-tab-panel': DsTabPanel }
}

```

The panel hides itself with `:host(:not([selected])) { display: none; }`. `display: none` also removes the content from the accessibility tree, which is exactly what you want for an inactive panel.

Step 3: The Parent and Its Slots

Now the coordinator, which is boxes 2 and 3 in Figure 1-1. Its public surface is three properties and two queries for its children:

Example 1-3. The public properties and slot queries of ds-tabs

```

/** Name of the selected panel. Defaults to the first enabled tab. */
@property({ reflect: true }) selected = '';
/** `auto` selects on focus; `manual` waits for Enter or Space. */
@property({ reflect: true }) activation: 'auto' | 'manual' = 'auto';
@property({ reflect: true }) orientation: 'horizontal' | 'vertical' = 'horizontal';

@queryAssignedElements({ slot: 'tab' }) private _tabs!: DsTab[];
@queryAssignedElements({ selector: 'ds-tab-panel' }) private _panels!: DsTabPanel[];

```

`@queryAssignedElements` returns the elements currently assigned to a slot. The first query asks the `tab` slot; the second asks the default slot, filtered to `ds-tab-panel` so stray text or other elements are ignored. These are getters: they read the live DOM each time you use them, so there is no list to keep up to date.

The template creates the two slots and the tab list around the first of them:

Example 1-4. The ds-tabs template

```
render() {
  return html`
    <div
      class="tablist"
      part="tablist"
      role="tablist"
      aria-orientation=${this.orientation}
      @keydown=${this.#onKeydown}
      @click=${this.#onClick}
      @focusin=${this.#onFocusin}
    >
      <slot name="tab" @slotchange=${this.#sync}></slot>
      <span class="indicator" part="indicator" aria-hidden="true"></span>
    </div>
    <div class="panels" part="panels">
      <slot @slotchange=${this.#sync}></slot>
    </div>
  `;
}
```

The tab list is a `<div role="tablist">` in the shadow DOM. The tabs themselves are light DOM children, projected into the tab slot. The browser builds the accessibility tree from the flattened tree, so for assistive technology the slotted tabs are children of the tablist.

The three event handlers are on the tab list, not on each tab. Clicks, key presses and focus changes from slotted tabs bubble up through the slot to the wrapper, so one listener serves every tab, including tabs added later. This is event delegation, and it is the reason the component needs no per-tab setup code.

The indicator `<span>` is decoration. It is `aria-hidden` so screen readers skip it.

Step 4: One Function That Makes Everything Agree

The most important design decision in the component is this. Rather than writing handlers that each update a little bit of state (“when a tab is clicked, mark it selected, unmark the old one, show its panel, hide the old one”), there is a single function that takes the current `selected` value and makes every child match it:

Example 1-5. The sync function

```
/** Make every tab and panel agree with `selected`. Safe to call repeatedly. */
#sync = () => {
  const tabs = this._tabs;
  const panels = this._panels;
  if (!tabs?.length) return;

  // Fall back to the first enabled tab when nothing valid is selected.
  const names = tabs.filter((t) => !t.disabled).map((t) => t.panel);
```

```

if (!names.includes(this.selected)) this.selected = names[0] ?? '';

// IDREFs only resolve within one tree. Tabs and panels are siblings in the
// light DOM, so the coordinator writes the ids as plain attributes.
for (const el of [...tabs, ...panels]) {
  if (!el.id) el.id = `ds-tabs-${++nextId}`;
}
for (const tab of tabs) {
  const panel = panels.find((p) => p.name === tab.panel);
  tab.selected = tab.panel === this.selected;
  if (panel) {
    tab.setAttribute('aria-controls', panel.id);
    panel.setAttribute('aria-labelledby', tab.id);
    panel.selected = tab.selected;
    panel.tabIndex = 0;
  }
}

this.#setActive(tabs.find((t) => t.selected) ?? tabs[0]);
this.#positionIndicator();
};

```

Read it top to bottom:

1. **Choose a valid selection.** If `selected` does not name an enabled tab (it was empty, misspelled, or points at a disabled tab), fall back to the first enabled tab.
2. **Give every element an id** if it does not have one. The ids are generated from a module-level counter.
3. **Link tabs to panels.** For each tab, find the panel with the matching name, then write `aria-controls` on the tab and `aria-labelledby` on the panel. Set `selected` on both, and give the panel `tabindex="0"`.
4. **Reset the roving `tabindex`** and move the indicator.

The function is safe to call as often as you like, and the component does: when `selected` changes, and whenever either slot reports that its children changed. That second trigger is what makes dynamic content work. Append a new tab and panel to a mounted `<ds-tabs>` and `slotchange` fires, the function runs, and the new pair is wired up. One of the tests in this chapter does exactly that.

Why the Parent Writes the ARIA IDs

`aria-controls` and `aria-labelledby` take element ids, and an id only resolves inside the tree where it lives. If the tabs lived inside a shadow root and the panels outside it, the references would silently point at nothing.

In this component the tabs and panels are siblings in the light DOM, so ids work. But the elements do not know about each other: each is its own custom element with its own shadow root. Only the parent sees both. So the parent writes the ids and the relationships as ordinary attributes on the light DOM elements. If you ever need the relationship across a shadow boundary, ids are the wrong tool, and you should reach for a different structure.

Step 5: Roving Tabindex

A browser moves focus to the next element in the tab order when a user presses Tab. If every tab had `tabindex="0"`, a list of ten tabs would take ten key presses to cross, which defeats the point of a widget with its own arrow-key navigation.

The standard solution is **roving tabindex**: exactly one tab has `tabindex="0"` and is in the page tab order; the rest have `tabindex="-1"`, meaning they can receive focus from code or a click but are skipped by Tab. The 0 rolls from tab to tab as focus moves.

Example 1-6. Roving tabindex

```
/** Roving tabindex: exactly one tab is in the page tab order. */
#setActive(active: DsTab | undefined) {
  this.#active = active;
  for (const tab of this._tabs) tab.tabIndex = tab === active ? 0 : -1;
}
```

Note that the active tab is tracked separately from the selected tab. In manual mode, arrow keys move focus without selecting, so the tab that holds `tabindex="0"` can differ from the one that is selected. When the user tabs away and later returns, focus lands on the last tab they were on. A `focusin` handler keeps `#active` current.

Step 6: The Keyboard

The key handler is a switch over `event.key`:

Example 1-7. The keyboard handler

```
#onKeydown(event: KeyboardEvent) {
  const current = this.#tabFrom(event);
  if (!current) return;

  const enabled = this._tabs.filter((t) => !t.disabled);
  const i = enabled.indexOf(current);
  const vertical = this.orientation === 'vertical';
  const next = vertical ? 'ArrowDown' : 'ArrowRight';
  const prev = vertical ? 'ArrowUp' : 'ArrowLeft';

  let target: DsTab | undefined;
  switch (event.key) {
    case next: target = enabled[(i + 1) % enabled.length]; break;
    case prev: target = enabled[(i - 1 + enabled.length) % enabled.length]; break;
    case 'Home': target = enabled[0]; break;
    case 'End': target = enabled[enabled.length - 1]; break;
    case 'Enter':
    case ' ':
      event.preventDefault();
      this.#select(current);
      return;
  }
```

```

    default: return;
  }

  event.preventDefault();
  if (!target) return;
  target.focus();
  if (this.activation === 'auto') this.#select(target);
}

```

A few details of the handler matter.

- **Disabled tabs are filtered out first.** Arrow keys walk the list of enabled tabs, so they step over disabled ones without any special case in the movement logic.
- **Wrapping is modular arithmetic.** `(i + 1) % length` wraps from the last tab to the first. The `- 1` case adds `length` first so the result never goes negative.
- **Orientation changes only which keys mean next and previous.** The movement logic is identical for horizontal and vertical.
- **Focus moves first, selection second.** In auto mode, `#select` runs after `target.focus()`. In manual mode it does not run at all until Enter or Space.

Warning: Call `preventDefault()` on handled keys. Space and the arrow keys scroll the page by default, so without it Down would move the selection and scroll the page at once.

Step 7: A Cancelable Event

When the user selects a tab, the parent announces it with an event before changing its own state:

Example 1-8. Selecting a tab and firing the event

```

#select(tab: DsTab) {
  if (tab.disabled || tab.panel === this.selected) return;
  const event = new CustomEvent<TabChangeDetail>('ds-tab-change', {
    detail: { selected: tab.panel, previous: this.selected },
    bubbles: true,
    composed: true,
    cancelable: true,
  });
  if (this.dispatchEvent(event)) this.selected = tab.panel;
}

```

`dispatchEvent` returns `false` if a listener called `preventDefault()`. The component only updates `selected` when the event was not canceled. That gives an application a veto: a form with unsaved changes can stop the user from leaving a tab, ask for confirmation, and set `selected` itself afterward.

Two choices here are deliberate. First, the event fires only for **user-initiated** changes. Second, the event is `composed` and `bubbles`, so it escapes shadow roots and can be handled far above the component.

Note: Setting `el.selected = 'b'` from code does not fire the event. The caller already knows what changed, and events for programmatic changes invite infinite loops.

The event's name carries a `ds-` prefix. Custom events share a global namespace with every other library on the page, and a bare `change` event would be indistinguishable from the native one.

Putting the last three steps together, Figure 1-3 traces one `ArrowRight` press from the keyboard to the updated DOM.

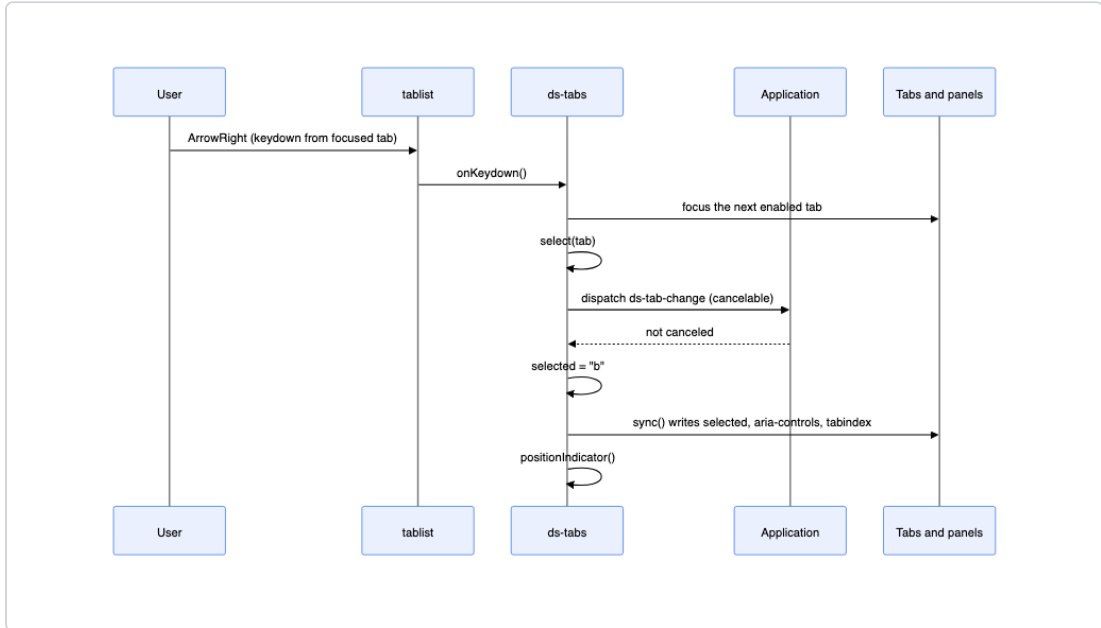


Figure 1-3. What happens when the user presses `ArrowRight` (automatic activation)

Step 8: The Sliding Indicator

The selected tab has an underline that slides from tab to tab. Browsers can animate `transform` and size changes, but they need numbers to animate between, so the component measures:

Example 1-9. Measuring the indicator position

```
#positionIndicator() {
  const list = this.renderRoot.querySelector<HTMLElement>('.tablist');
  const tab = this._tabs?.find((t) => t.selected);
  if (!list || !tab) return;
  const l = list.getBoundingClientRect();
  const t = tab.getBoundingClientRect();
  list.style.setProperty('--x', `${t.left - l.left + list.scrollLeft}px`);
  list.style.setProperty('--y', `${t.top - l.top + list.scrollTop}px`);
  list.style.setProperty('--w', `${t.width}px`);
  list.style.setProperty('--h', `${t.height}px`);
  // Enable the slide transition only after the first placement.
  requestAnimationFrame(() => list.toggleAttribute('data-ready', true));
}
```

The method compares the bounding rectangles of the selected tab and the tab list and stores the difference in four CSS custom properties. The stylesheet reads them:

Example 1-10. Indicator styles

```
.tablist {
  position: relative;
  display: flex;
  overflow: auto;
  border-block-end: var(--sem-border-width) solid var(--sem-color-border);
}
:host([orientation='vertical']) .tablist {
  flex-direction: column;
  flex: none;
  border-block-end: 0;
  border-inline-end: var(--sem-border-width) solid var(--sem-color-border);
}
.indicator {
  position: absolute;
  inset-block-end: 0;
  inset-inline-start: 0;
  width: var(--w, 0);
  height: var(--indicator-size);
  background: var(--indicator-color);
  transform: translate(var(--x, 0), 0);
  pointer-events: none;
}
```

Why `getBoundingClientRect` and not `offsetLeft`? `offsetLeft` is relative to an `offsetParent`, and for slotted elements that parent can be retargeted across the shadow boundary, which makes the number hard to reason about. Rectangles are in viewport coordinates, so subtracting two of them gives the right answer regardless of where the elements sit in the tree. Adding `scrollLeft` accounts for a tab list that has scrolled because it overflowed.

The `data-ready` attribute fixes a small visual bug. Without it, the indicator would slide in from the top-left corner on first render, because its starting position is zero. The transition is only enabled after the first placement is done. The tab list also re-measures whenever the component's own size changes, through a `ResizeObserver` created in the field initializer and attached in `connectedCallback`.

Seeing the Component Run

The screenshots in this section are real. A script starts the demo page, opens it in Google Chrome with a clean profile, drives the component with real mouse and keyboard input through Playwright, and captures the Chrome window itself. Nothing here is a mockup. To reproduce them on macOS, run `npm run capture` from the book's repository.

Figure 1-4 shows automatic activation after a click on Specs. The indicator sits under the selected tab, the selected label takes the action color, and Reviews is dimmed because it is disabled: arrow keys skip it and a click on it does nothing.

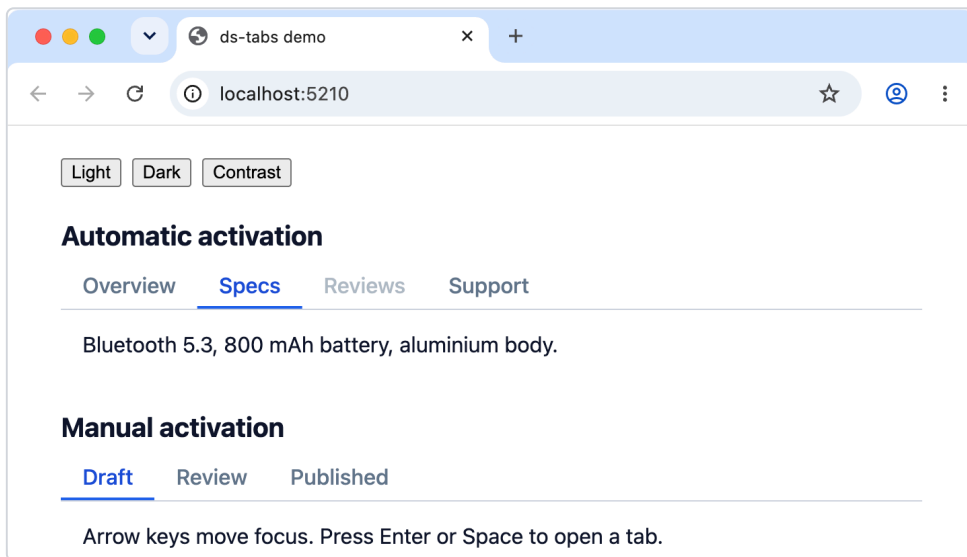


Figure 1-4. Automatic activation, with the second tab selected and the third tab disabled

Figure 1-5 shows keyboard focus. The script pressed Tab until focus reached the tab list, then ArrowRight. Focus and selection both moved to Specs, and the focus ring appeared around it. The ring is drawn with `:focus-visible`, which the browser applies only when it believes the user is navigating by keyboard. That is

why the mouse click in Figure 1-4 shows no ring.

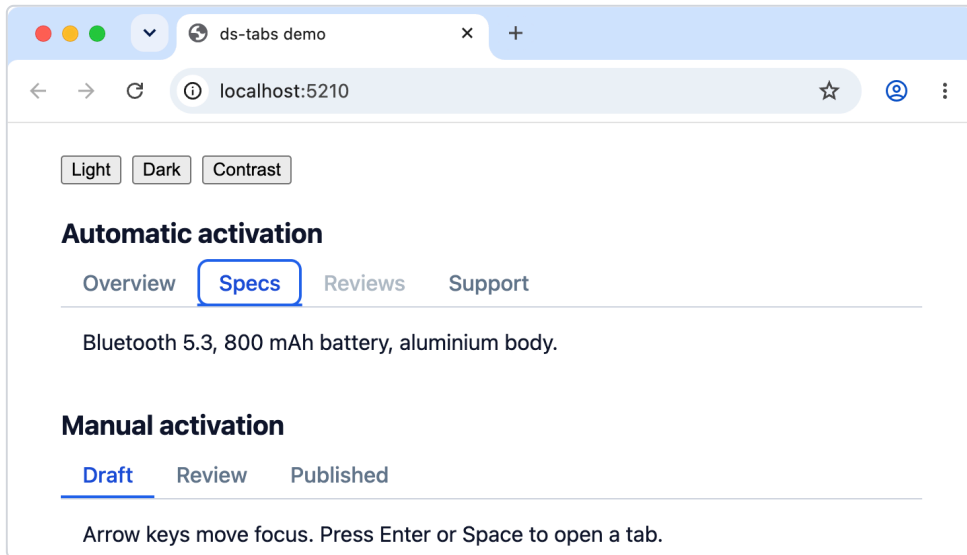


Figure 1-5. Keyboard focus on the selected tab

Figure 1-6 shows manual activation. Focus is on Review, as the ring shows, but Draft is still the selected tab and its panel is still showing. The selection will change only when the user presses Enter or Space.

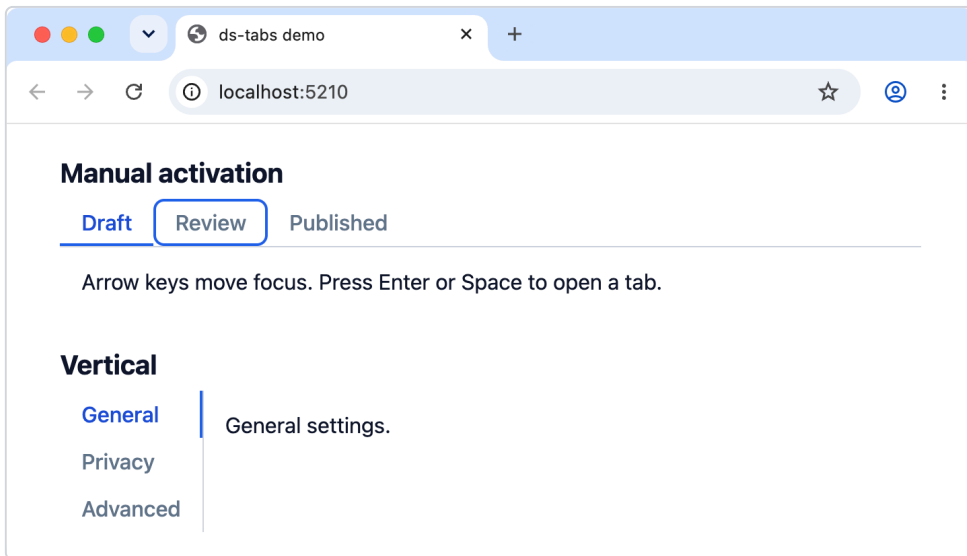


Figure 1-6. Manual activation: focus and selection are different tabs

Figure 1-7 shows the vertical orientation, in which the indicator moves to the inline-end edge of the list and Up and Down replace Left and Right.

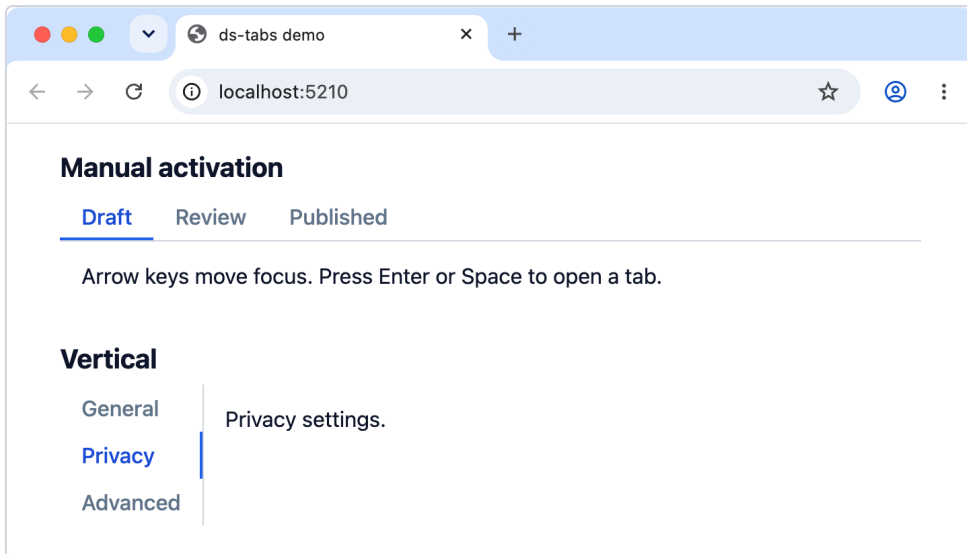


Figure 1-7. Vertical orientation

Figure 1-8 shows the same component with the dark theme applied. The component code and the markup are unchanged. The only difference is `data-theme="dark"` on the document element, which remaps the semantic tokens that the component reads.

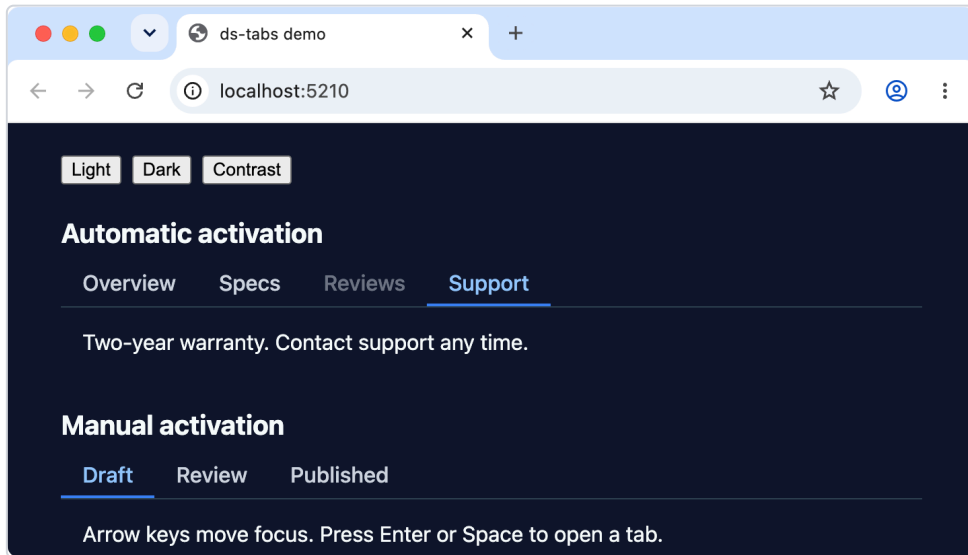


Figure 1-8. The same component in the dark theme

Inspecting the Shadow DOM in Chrome DevTools

Everything described in the concepts section is visible in the browser's own tools. Open Chrome DevTools (on macOS, Command-Option-I), choose the Elements panel and select `<ds-tabs>`. Figure 1-9 is a capture of that panel from the running component, with the `#shadow-root` expanded.

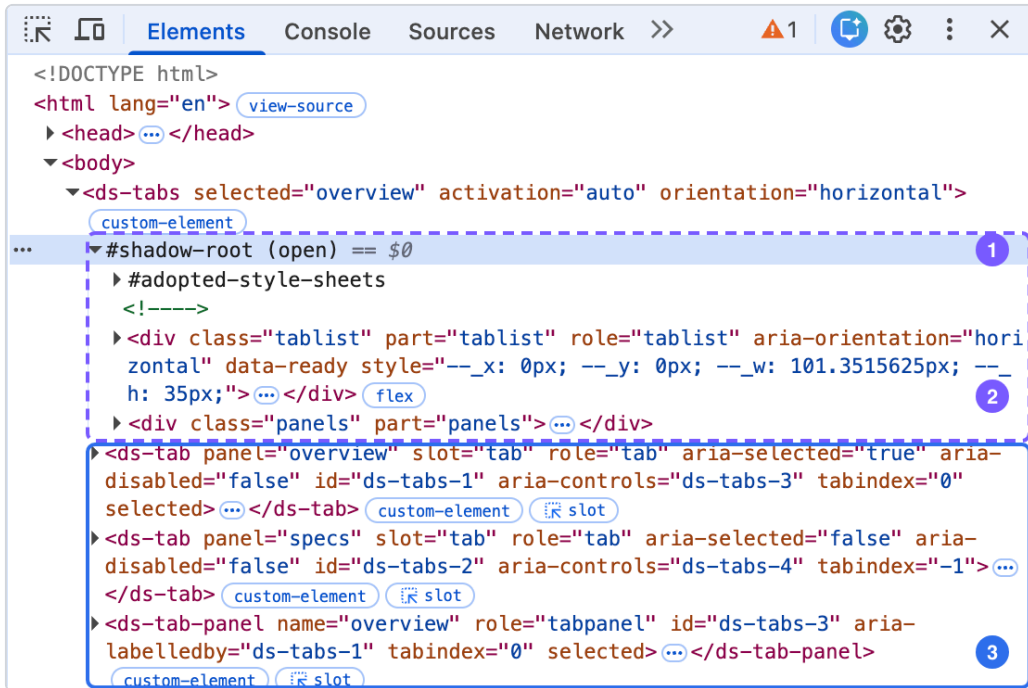


Figure 1-9. The shadow root and the light DOM of ds-tabs in Chrome DevTools

Read the numbered callouts from the top:

1. **#shadow-root (open)** is the shadow tree that Lit attached to `<ds-tabs>`. It is the selected row, and the dashed outline covers everything inside it. DevTools labels it *open*, which is why `host.shadowRoot` works from code. The `#adopted-style-sheets` entry inside it holds the component's CSS, and the empty comment is a marker Lit leaves for its template.
2. **div.tablist** and **div.panels** are the two elements that `render()` creates. They have `part` attributes, which is what lets a page style them with `::part()`. Expand them and you find the two `<slot>` elements and the indicator.
3. **ds-tab** and **ds-tab-panel** sit below the shadow root with a solid outline. They are *light DOM* children, siblings of `#shadow-root` and not part of it. The `slot` badge that DevTools draws next to each one marks it as assigned to a slot in the shadow tree.

The attributes on those light DOM rows are the ones the component wrote at runtime. Each `ds-tab` has a `role`, `aria-selected`, `tabindex`, an `id` and an `aria-controls` that points at its panel. Each `ds-tab-panel` has the matching `aria-labelledby`. None of them appear in the markup you author; they are the output of `sync()` from Step 4.

Tip: Selecting an element in the Elements panel makes it available in the Console as `$0`. Type `$0.assignedElements()` after selecting a `<slot>` to see which light DOM nodes it holds.

What the Browser Exposes to Assistive Technology

A screenshot cannot show the part of a tabs component that matters most to many users. Playwright can report the accessibility tree the browser builds for an element, in the same shape that assistive technology receives. This is the real output for the first tabs on the demo page:

Example 1-11. The accessibility tree of the first tabs component

```
- tablist:
  - tab "Overview" [selected]
  - tab "Specs"
  - tab "Reviews" [disabled]
  - tab "Support"
- tabpanel "Overview": A compact, quiet keyboard with a long battery life.
```

Read it as a screen reader would. There is a `tablist` containing four tabs. Overview is marked selected and Reviews is marked disabled. Only one `tabpanel` appears, because the inactive panels use `display: none` and are removed from the tree. The panel's name is "Overview", which it received from its tab through `aria-labelledby`. That one line is the evidence that the ID wiring from Step 4 works: the tab and the panel are siblings in the light DOM, so the ids resolve.

Tip: Run this snapshot against your own components while you develop them. When a role, a name or a state is missing from the output, it is missing for screen reader users too.

Styling API

Everything a consumer can customize, in one place.

Tokens consumed (the semantic layer):

Semantic token	Used for
<code>--sem-color-text-muted</code>	unselected tab label
<code>--sem-color-action-subtle-text</code>	selected tab label
<code>--sem-color-action-bg</code>	indicator
<code>--sem-color-border</code>	tab list divider
<code>--sem-color-focus-ring</code> , <code>--sem-focus-ring-width</code>	focus outline
<code>--sem-motion-duration</code> , <code>--sem-motion-easing</code>	indicator and color transitions

Component tokens (override on one instance):

Token	Default
<code>--ds-tab-fg</code>	muted text
<code>--ds-tab-fg-selected</code>	subtle action text
<code>--ds-tabs-indicator-color</code>	action background
<code>--ds-tabs-indicator-size</code>	focus ring width
<code>--ds-tabs-panel-padding</code>	<code>--sem-space-stack-md</code>

Parts (`::part()`): `tablist`, `indicator`, `panels`.

Reduced motion: the semantic layer sets the motion durations to zero under `prefers-reduced-motion`: reduce, so the indicator jumps instead of sliding. The component does not need its own media query.

From the outside, a themed instance looks like this:

Example 1-12. Theming a single instance from outside

```
ds-tabs {
  --ds-tabs-indicator-color: tomato;
  --ds-tab-fg-selected: tomato;
}
ds-tabs::part(tablist) {
  border-block-end-style: dashed;
}
```

Accessibility

What the component provides: roles on all three elements, `aria-selected` and `aria-disabled` on tabs, `aria-controls` and `aria-labelledby` by linking each pair, `aria-orientation` on the tab list, a single tab stop, and a visible focus ring that works in the dark and high-contrast themes because it reads semantic tokens.

What it does not provide, and what the application is still responsible for: a name for the tab list when a page has several of them, and sensible content in each panel. The tab list lives inside the shadow root, so an `aria-label` on `<ds-tabs>` does not reach it; the component would have to forward the label, which is exercise 3.

A Correction Worth Keeping

A first version of this component set the roles with `ElementInternals`: `this.#internals.role = 'tab'` and `ariaSelected`. That is attractive, because it gives the element default semantics without touching its attributes. The automated test then failed:

```
Rule: aria-required-children
Element has children which are not allowed: ds-tab[aria-controls]
```

axe-core read the tab as an element with no role at all, so the `tablist` appeared to have no valid children. The test measures what the tool can see, not what a screen reader announces, and the two can differ. But a component whose accessibility cannot be checked by the standard tooling is hard to trust and hard to maintain, so the roles became plain host attributes, which every tool reads. The failure also exposed a second finding, a color-contrast complaint on the disabled tab, which went away once `aria-disabled` was a real attribute.

Note: Use `ElementInternals` for form participation (Chapter 3), where it is the only option. For ARIA on the host, attributes are the more portable choice today.

Testing

The test file builds one reusable fixture and drives the component the way a user would. The keyboard test uses `sendKeys`, which sends real key presses through the browser's automation protocol rather than dispatching synthetic events:

Example 1-13. A keyboard test using real key presses

```
it('moves with arrow keys, skips disabled, and wraps (auto activation)', async () => {
  const el = await fixture<DsTabs>(basic());
  await settle(el);
  tabs(el)[0].focus();
  await sendKeys({ press: 'ArrowRight' });
  expect(el.selected).to.equal('b');
  await sendKeys({ press: 'ArrowRight' });
  expect(el.selected).to.equal('d'); // skipped disabled "c"
  await sendKeys({ press: 'ArrowRight' });
  expect(el.selected).to.equal('a'); // wrapped
  await sendKeys({ press: 'ArrowLeft' });
  expect(el.selected).to.equal('d');
  await sendKeys({ press: 'Home' });
  expect(el.selected).to.equal('a');
  await sendKeys({ press: 'End' });
  expect(el.selected).to.equal('d');
  expect(document.activeElement).to.equal(tabs(el)[3]);
});
```

The suite covers the default selection, ARIA wiring, click selection, disabled tabs, arrow-key wrapping, Home and End, manual activation, the single tab stop, vertical orientation, the cancelable event, programmatic changes, tabs added after first render, an unknown `selected` value, and an axe audit.

Running the Suite

Run the suite from the package folder:

```
npm test
```

Web Test Runner starts a development server, opens each test file in Chromium through Playwright, runs the tests and reports the results. Example 1-14 is the real output from the run used for this chapter.

Example 1-14. Output of `npm test`

```
$ npm test

ds-tabs [Chromium]
  ✓ selects the first enabled tab by default
  ✓ wires roles, ids and ARIA relationships
  ✓ selects on click and ignores disabled tabs
  ✓ moves with arrow keys, skips disabled, and wraps (auto activation)
  ✓ manual activation moves focus but selects only on Enter or Space
  ✓ keeps a single tab stop (roving tabindex) while focus moves in manual mode
  ✓ uses Up/Down arrows when vertical
  ✓ fires a cancelable ds-tab-change event
  ✓ does not fire the event for programmatic changes
  ✓ wires tabs added after first render
  ✓ falls back to the first enabled tab for an unknown selected value
  ✓ is accessible
🚧 Browser logs:
  Lit is in dev mode. Not recommended for production! See https://lit.dev/msg/dev-mode for
  more information.
Finished running tests in 1s, all tests passed! 🎉
```

Each check mark is a passing test, and all twelve passed. The runner exits with status 0 when everything passes, which is what lets a continuous integration job gate a merge on it. The “Lit is in dev mode” message is Lit’s standard notice when it runs without a production build, and it is harmless in tests.

The test names read as a specification: “moves with arrow keys, skips disabled, and wraps” says what the component promises, without mentioning any internal function. That is deliberate. If you later rewrite the internals, the names, and the tests under them, should not need to change.

Tip: When a test fails, the runner prints the failing assertion with the expected and actual values and the line in the test file. Add `--watch` to `wtr` to rerun the suite whenever a file changes while you work.

Tip: `sendKeys` acts on whatever has focus, so begin every keyboard test by focusing a tab. If a test passes alone but fails in the suite, an earlier test left focus elsewhere.

Pitfalls

Setting `selected` before the children exist. If `<ds-tabs selected="b">` is parsed before its tabs are, there is nothing to validate against yet. The component handles this by re-running the `sync` function on `slotchange`, so the value is validated once the children arrive. If you replace this with a one-time check in `firstUpdated`, it will break for markup that is built dynamically.

Assuming slot content is ready in `connectedCallback`. Slot assignment happens after render. Reading `_tabs` that early returns an empty array. Do slot-dependent work in `slotchange` or `updated`.

Mutating attributes you also observe. The `sync` function writes `selected` on children, and a child's `selected` property is reflected. If the parent also *listened* to those attributes, you would have a loop. The parent only reads its own `selected` and writes the children's.

Duplicate ids. If an author gives two tabs the same id, the ARIA links break silently. The component only fills in ids that are missing and never rewrites existing ones, so duplicate ids are the author's problem. A development-mode warning would be a good addition.

Right-to-left layouts. Arrow keys in a horizontal tab list should be mirrored in right-to-left documents, and the indicator's horizontal offset should be computed from the inline start edge. This version ignores both. It is exercise 4.

Summary

You built a three-element compound component. The pieces worth carrying forward:

- A parent finds its children with named slots and `@queryAssignedElements`, and re-synchronizes on `slotchange`.
- One idempotent function reconciles all child state with one source of truth, instead of many handlers each updating a little.
- ARIA id relationships are written by the parent as attributes, because ids only resolve within a single tree.
- Roving `tabindex` makes a widget one tab stop, and the active tab is tracked separately from the selected one.
- User-initiated changes fire a cancelable, composed, prefixed event; programmatic changes do not.
- Real key presses and an automated accessibility audit catch problems a visual check does not.

The next chapter keeps the focus on browser-managed behavior but raises the stakes: dialogs, popovers and tooltips live in layers above the page, steal and return focus, and have to position themselves relative to other elements.

Exercises

Each exercise extends this chapter's component. Add a test where the behavior can be tested. Worked solutions, each with a test, are in section E.3 of Appendix E and in `packages/tabs/solutions/`.

1. **Accordion.** Build `<ds-accordion>` and `<ds-accordion-item>` using the same parent-and-children structure. Each item has a header button and a panel. Support a `multiple` attribute that allows several items open at once. Keyboard: Up and Down move between headers, Home and End jump to the ends. Reuse the single-sync-function approach from Step 4.
2. **Closeable tabs.** Add a `closable` attribute to `<ds-tab>` that shows a close button. Closing the selected tab should select its neighbor. Fire a cancelable `ds-tab-close` event first.
3. **Label forwarding.** Let the author name the tab list with `<ds-tabs label="Product details">` and forward it to the tablist as `aria-label`. Add a test that asserts it.

4. **Right-to-left.** Make arrow keys and the indicator respect `dir="rtl"`. Add a test that sets the direction on the element and presses `ArrowRight`.
5. **Lazy panels.** Add a `lazy` attribute to `<ds-tab-panel>` so its content is rendered only the first time the panel is selected. Think about what should happen to the content when the user leaves the tab.



PART II

Forms and Input



4 Combobox

Chapter 4. Combobox

A combobox is a text box that suggests answers. The user types a few letters, a list of matches appears, and they either keep typing or pick one. You have used one every time a search box offered completions, a form offered a list of countries, or an address field finished a street name for you.

It is also one of the hardest widgets to get right. The visual idea is simple, but the interaction has to work for a person who cannot see the list at all. A screen reader user needs to hear that a list exists, how many results it holds, which one is highlighted and what happens when they press Enter, all while keyboard focus stays in the text box where they are typing. The WAI-ARIA Authoring Practices Guide describes this pattern in some detail, and this chapter implements it.

The second half of the problem is data. Real comboboxes often fetch their suggestions from a server, which means requests that take time, can fail, and can arrive out of order. Handling that cleanly is as much a part of the component as the markup.

In this chapter, you will learn:

- how the ARIA combobox pattern keeps focus in the input while a list is navigated
- how `aria-activedescendant`, `aria-expanded` and `aria-controls` express the state of a popup list
- how to announce results to assistive technology with a live region
- how to use Lit's `Task` to load options asynchronously, with loading, empty and error states
- how to cancel stale requests with an `AbortSignal` and avoid out-of-order responses
- how to debounce typing, highlight matches safely, and handle text composed with an input method

What We Are Building, and Why

`ds-combobox` is an **editable combobox with list autocomplete**. The user can type free text, the list filters to matching options, and choosing one sets the component's value. It works with a fixed list of options, or with a function that fetches options for each query.

Figure 4-1 shows the structure the component must expose. There is an input with the `combobox` role, a list with the `listbox` role containing `option` elements, and a visually hidden status region. They all live in one shadow root, which matters, because the attributes that connect them are references by `id`, and `id` references only work inside a single tree.

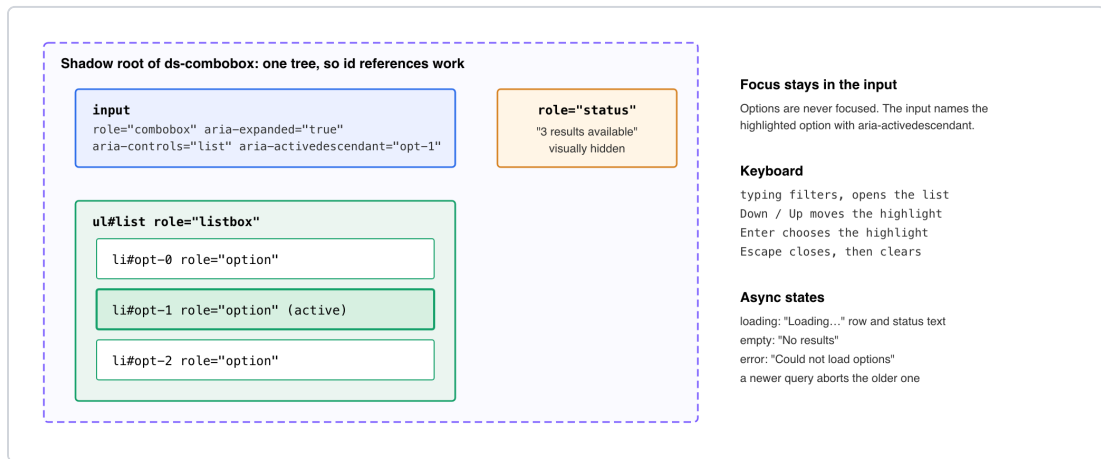


Figure 4-1. The structure of ds-combobox

Figure 4-2 shows the asynchronous half. Each keystroke restarts a short timer. When typing pauses, the search value changes, and a Lit Task calls the loader. If the user types again before the answer arrives, the Task cancels the old request and starts a new one, so only the answer to the latest query is ever shown.

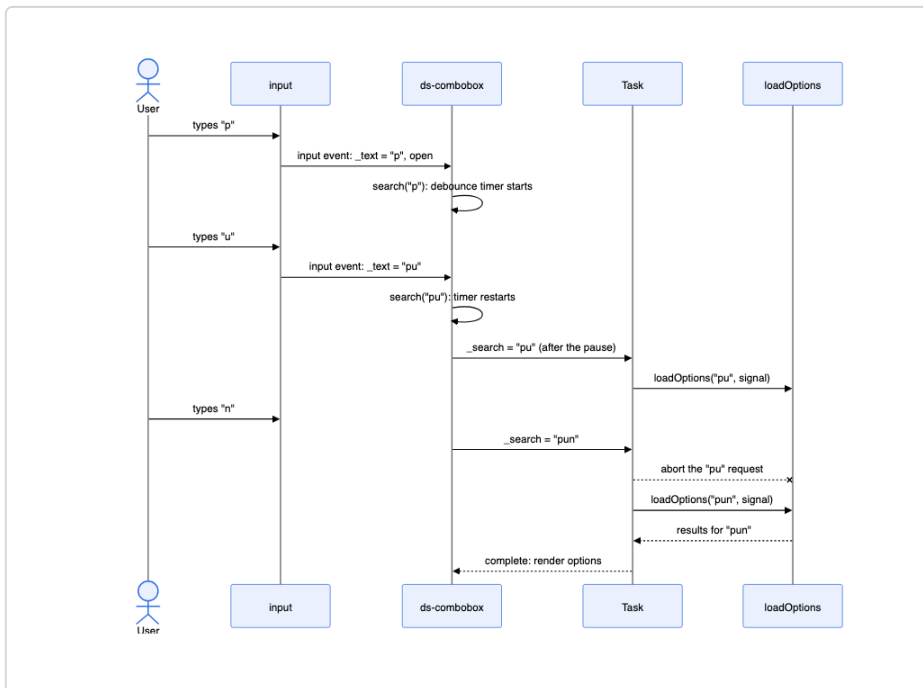


Figure 4-2. Searching asynchronously

Technical Requirements

You need Node 20 or newer and a current version of Google Chrome. The package uses `@lit/task`, which is listed in its `package.json`.

```

cd packages/combobox
npm install
npm run dev      # a static combobox and one with a simulated slow server
npm test         # 21 tests
  
```

Concepts You Need Before Writing Code

The combobox draws on two groups of ideas. The first is the ARIA pattern, with its key attribute and its live region. The second is asynchronous data: a Lit `Task`, cancellation and debouncing.

The Combobox Pattern

The ARIA combobox pattern has a few firm rules.

1. The text input has `role="combobox"`.
2. The input has `aria-expanded`, which is `true` while the popup is showing and `false` otherwise.
3. The input has `aria-controls`, whose value is the `id` of the popup.
4. The popup has `role="listbox"` and holds elements with `role="option"`.
5. **Focus never moves into the popup.** It stays in the input, so the user can keep typing. Instead, the input's `aria-activedescendant` holds the `id` of the highlighted option, and the browser tells assistive technology to treat that option as if it had focus.
6. `aria-autocomplete="list"` tells the user that the list filters what they type.

The keyboard model follows from rule 5. Down and Up arrows move the highlight, Enter chooses the highlighted option, and Escape closes the list. A second Escape clears the text.

`aria-activedescendant`

Normally, assistive technology follows keyboard focus. When an option in a list is “active” without actually holding focus, the page needs another way to say which one it is. `aria-activedescendant` is that way: it names a descendant by `id`, and the browser exposes it as the focused element for the purposes of a screen reader.

This is the reason the input, listbox and options must share a tree. If the list were in a different shadow root from the input, the `id` would not resolve and the highlight would be silent.

Live Regions

A sighted user sees a list appear. A screen reader user does not, unless something tells them. A **live region**, an element with `role="status"` or `aria-live="polite"`, makes a screen reader announce changes to its text content. The component keeps a visually hidden status element and writes to it: “3 results available”, “Loading”, “No results”, “Could not load options”. Hiding it visually, with a `clip` rather than `display: none`, keeps it in the accessibility tree.

Lit's Task

A Task from `@lit/task` is a reactive controller that runs an asynchronous function whenever its arguments change, and keeps track of the state of the latest run. You give it two functions:

```
new Task(this, {
  args: () => [this.query] as const,
  task: async ([query], { signal }) => fetchOptions(query, signal),
});
```

`args` returns the values the task depends on. When any of them changes between updates, the task runs again. `task` does the work and receives an `AbortSignal`. The task exposes a `status` (`INITIAL`, `PENDING`, `COMPLETE` or `ERROR`), the latest `value` and `error`, and a `render()` helper that takes one function per state.

Two properties make it the right tool for this component. The `signal` is aborted when a newer run starts, so the old request can be canceled. And the task only ever records the result of the **latest** run, so a slow response to an old query cannot overwrite a fast response to a new one.

Canceling Requests

An `AbortController` produces an `AbortSignal`. Passing the signal to `fetch(url, { signal })` makes the request fail with an `AbortError` as soon as the controller aborts. For other asynchronous work, a function can listen for the signal's `abort` event and clean up, for example by clearing a timer.

Debouncing

Typing “pune” produces four input events in a fraction of a second. Sending four requests would be wasteful and, because responses can come back in any order, risky. **Debouncing** waits for a pause: each keystroke restarts a timer, and only when the timer expires is the search performed. A delay of around 200 milliseconds is a common starting point.

Step 1: The Structure

The render method builds everything Figure 4-1 shows.

Example 4-1. Rendering the combobox

```
render() {
  const listBoxId = `${this.#id}-list`;
  const options = this.#task.value ?? [];
  const showPopup = this._open && !this.disabled;
  return html`
    ${this.label ? html`<label part="label" for="input">${this.label}</label>` : nothing}
    <div class="wrap">
      <input
        id="input"
        part="input"
        type="text"
        role="combobox"
        autocomplete="off"
        aria-autocomplete="list"
        aria-expanded=${showPopup ? 'true' : 'false'}
        aria-controls=${listBoxId}
        aria-activedescendant=${ifDefined(showPopup && this._active >= 0 ? `${this.#id}-
opt-${this._active}` : undefined)}
        aria-required=${this.required ? 'true' : 'false'}
        placeholder=${ifDefined(this.placeholder || undefined)}
        ?disabled=${this.disabled}
        .value=${this._text}
        @input=${this.#onInput}
        @keydown=${this.#onKeydown}
        @focus=${() => !this.#selected && this.#search('')}
        @blur=${this.#onBlur}
      >
```

```

    />
    ${this._text && !this.disabled
      ? html`<button class="clear" type="button" aria-label="Clear" @mousedown=${(e:
Event) => e.preventDefault()} @click=${this.#clear}>&#x2715;</button>`
      : nothing}
    <ul id=${{listboxId}} part="listbox" role="listbox" aria-label=${this.label ||
'Suggestions'} ?hidden=${!showPopup} @mousedown=${(e: Event) => e.preventDefault()}>
      ${showPopup ? this.#renderBody() : nothing}
    </ul>
  </div>
  <div class="sr-only" role="status" aria-live="polite">${showPopup ?
this.#status(options) : ''}</div>
  `;
}

```

How the Code Works

The input carries the combobox attributes. `aria-expanded` and `aria-activedescendant` are bound to component state, and the `aria-controls` value, the list's id, is generated per instance from a counter, so two comboboxes on a page do not share ids.

The listbox is rendered even when it is closed, with the `hidden` attribute, so that `aria-controls` always points at an element that exists. Inside it, `#renderBody` renders either the options or a message.

Two small handlers keep the interaction smooth. The `mousedown` handler on the listbox and the clear button calls `preventDefault()`. A mouse press normally moves focus to the element under the pointer, which would take focus out of the input and fire `blur`, closing the popup before the click lands. Preventing the default of `mousedown` keeps focus in place, and the later `click` does the choosing.

The `@focus` handler starts an initial search when nothing is selected, so a loading list can be ready by the time the user starts typing.

Step 2: Loading Options with a Task

The Task is declared as a field, which makes it attach itself to the component.

Example 4-2. The options task

```

#task = new Task(this, {
  args: () => [this._search] as const,
  task: async ([query], { signal }) => {
    if (this.loadOptions) return this.loadOptions(query, signal);
    const q = query.trim().toLowerCase();
    return this.options.filter((o) => o.label.toLowerCase().includes(q));
  },
});

```

The task depends on `_search`, a piece of reactive state. It does not depend directly on what is in the text box. That indirection is what makes debouncing possible: the text box updates on every keystroke, while `_search`, the value the task watches, updates only after a pause.

If the application supplied a `loadOptions` function, the task calls it with the query and the signal. Otherwise it filters the static `options` list by a case-insensitive substring match. Both paths return a promise of options, so the rest of the component does not need to know which it is.

The three states are rendered through the task's helper:

Example 4-3. Rendering loading, empty, error and complete states

```
#renderBody() {
  return this.#task.render({
    initial: () => nothing,
    // Keep showing the previous results while loading, but only if there are some: an
    // earlier empty list is not worth keeping.
    pending: () => (this.#task.value?.length ? this.#renderOptions(this.#task.value) :
html`<li role="presentation" class="message">Loading...</li>`),
    complete: (list) => (list.length ? this.#renderOptions(list) : html`<li
role="presentation" class="message">${this.emptyText}</li>`),
    error: () => html`<li role="presentation" class="message error">Could not load options.
</li>`,
  });
}

#renderOptions(list: ComboboxOption[]) {
  const q = this._text.trim();
  return list.map(
    (o, i) => html`
    <li
      id="${this.#id}-opt-${i}"
      part="option"
      role="option"
      aria-selected=${o.value === this.value ? 'true' : 'false'}
      ?data-active=${i === this._active}
      @click=${() => this.#choose(o)}
    >${this.#highlight(o.label, q)}</li>
    `,
  );
}

/** Wrap the matching part of a label in <mark>. Lit escapes the text, so labels cannot
inject markup. */
#highlight(label: string, q: string) {
  const at = q ? label.toLowerCase().indexOf(q.toLowerCase()) : -1;
  if (at < 0) return label;
  return html`${label.slice(0, at)}<mark>${label.slice(at, at + q.length)}
</mark>${label.slice(at + q.length)}`;
}
```

While a task is pending, the component keeps showing the previous results if there are any, and a “Loading...” row if there are none. Showing the old list while the new one loads avoids a flicker of an empty popup on every keystroke. A `complete` result with no items shows the empty text, and an `error` shows a plain message.

Highlighting Safely

`#highlight` finds the typed text inside each label and wraps it in `<mark>`. It builds a Lit template from three pieces of the label, not a string of HTML, so Lit escapes each piece. A label that contains markup characters is displayed as text and cannot inject elements. This is the reason to avoid assembling HTML strings for anything that comes from data.

Step 3: Keyboard and Selection

The input's handlers implement the keyboard model, the debounce and the selection.

Example 4-4. Searching, typing and keys

```

/** Queue a search, waiting for a pause in typing so that each keystroke does not start a request. */
#search(query: string) {
  clearTimeout(this.#timer);
  const run = () => { this._search = query; };
  if (this.debounce > 0) this.#timer = window.setTimeout(run, this.debounce);
  else run();
}

#onInput(event: Event) {
  this._text = (event.target as HTMLInputElement).value;
  this._open = true;
  this._active = -1;
  this.#search(this._text);
}

/** Enter with nothing to choose submits the form, as it does in a native text input. */
#submitOnEnter(event: KeyboardEvent) {
  // A native text input submits its form when Enter is pressed. An input inside a shadow root is
  // not a form control to the form, so the component has to do it: press the form's first submit button.
  if (event.key !== 'Enter' || event.isComposing || event.defaultPrevented) return;
  const form = this.#internals.form;
  const button = form && Array.from(form.elements).find(
    (el): el is HTMLButtonElement | HTMLInputElement => (el instanceof HTMLButtonElement ||
    el instanceof HTMLInputElement) && el.type === 'submit',
  );
  if (form && button && !button.disabled) form.requestSubmit(button);
}

#onKeyDown(event: KeyboardEvent) {
  if (event.isComposing) return; // an input method is mid-composition: leave the keys to it
  const options = this.#task.value ?? [];
  switch (event.key) {
    case 'ArrowDown':
      event.preventDefault();
      if (!this._open) { this._open = true; this.#search(this._text); }
  }
}

```

```

        this._active = options.length ? (this._active + 1) % options.length : -1;
        break;
    case 'ArrowUp':
        event.preventDefault();
        if (!this._open) { this._open = true; this.#search(this._text); }
        this._active = options.length ? (this._active <= 0 ? options.length - 1 : this._active
- 1) : -1;
        break;
    case 'Enter':
        if (this._open && this._active >= 0 && options[this._active]) {
            event.preventDefault();
            this.#choose(options[this._active]);
        } else {
            this.#submitOnEnter(event);
        }
        break;
    case 'Escape':
        if (this._open) {
            event.preventDefault();
            this._open = false;
            this._active = -1;
        } else if (this._text) {
            event.preventDefault();
            this.#clear();
        }
        break;
    default:
}
}
}

```

`#search` is the debounce. It clears any pending timer and either runs at once, when the debounce delay is zero, or schedules a run. Setting `_search` is what triggers the task.

`#onKeydown` begins with a guard: `if (event.isComposing) return;`. Some languages are typed through an **input method editor**, in which several keystrokes compose one character and Enter confirms the composition. If the component treated that Enter as “choose the highlighted option”, it would break typing in those languages. The browser flags such events with `isComposing`, and the component leaves them alone.

The arrow handlers use modular arithmetic to wrap around the list, and call `preventDefault()` so that the arrows do not also move the text cursor. Escape has two meanings, which depend on state: close the list if it is open, otherwise clear the text. Enter chooses the active option only when the list is open and something is highlighted. Otherwise the component submits the form itself, as a native text input would: it finds the form’s first enabled submit button and calls `form.requestSubmit(button)`. An input inside a shadow root is not a control of the form, so without this step Enter would do nothing, and a test that presses Enter in a form confirmed that.

Choosing an option updates three things at once, and announces it:

Example 4-5. Choosing an option

```
#choose(option: ComboboxOption) {
  this.#selected = option;
  this.value = option.value;
  this._text = option.label;
  this._open = false;
  this._active = -1;
  this.dispatchEvent(new CustomEvent('ds-select', { detail: { option }, bubbles: true,
composed: true }));
  this.dispatchEvent(new Event('change', { bubbles: true }));
}
```

The component fires a `ds-select` event carrying the whole option, and a plain `change` event for code that treats it like any other form control.

Blur and Restoring the Label

When focus leaves, the popup closes and the input text goes back to the label of the current selection. Without that step a user could type “zzz”, click away, and leave the box showing text that does not match the value the form will submit. Reverting to the selected label keeps what the box shows and what the form submits the same.

Step 4: Taking Part in a Form

This component implements form participation directly, without a mixin, so that it can be read on its own. It sets `static formAssociated = true`, calls `setFormValue` with the selected value, and reports `valueMissing` when it is required and nothing is chosen. It restores the initial value on `formResetCallback`. The details are the ones from Chapter 3, and the code is about twenty lines.

Note: A combobox has two “values”: the text in the box and the chosen option. Only the chosen option’s value goes to the form. Free text that matches nothing is not submitted.

Seeing the Component Run

The screenshots in this section are real, made in the same way as those in Chapter 1: a script opens the demo page in Google Chrome with a clean profile, drives the component with real mouse and keyboard input through Playwright, and captures the Chrome window. Nothing here is a mockup. To reproduce them on macOS, run `node scripts/capture-chapter.mjs ch04` from the book’s repository.

Figure 4-3 is the static combobox after typing the letter a and pressing ArrowDown twice. The list is filtered, and the active option is highlighted while keyboard focus stays in the input.

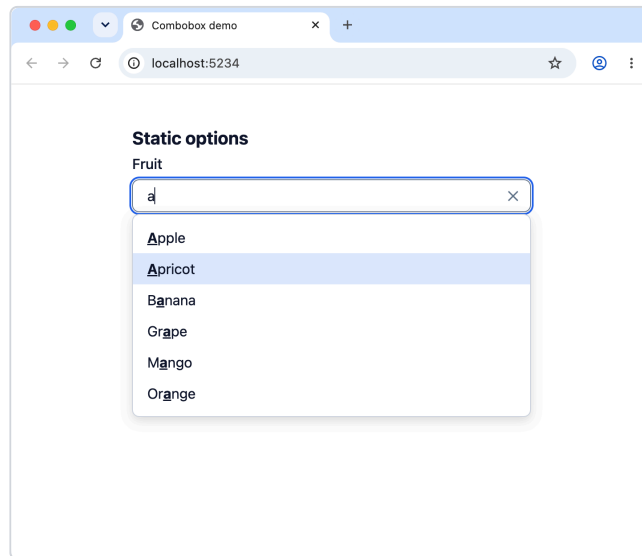


Figure 4-3. A filtered list with the second option active

Figure 4-4 is the asynchronous combobox while its request is in flight. The script made the server wait eight seconds so that the loading state could be captured, and the list shows a loading row in the meantime.

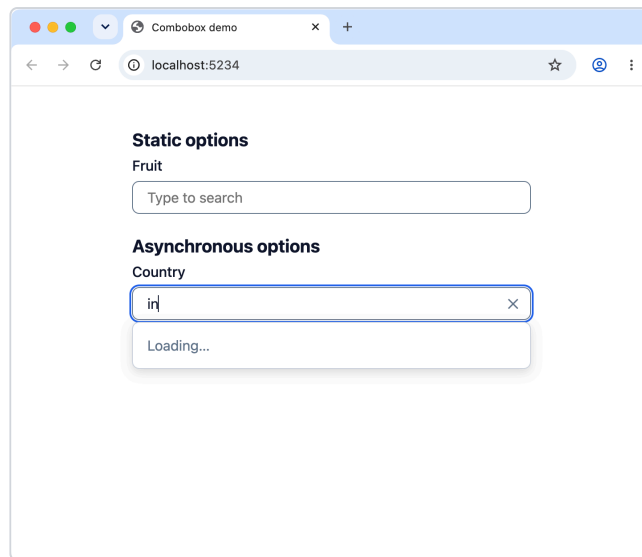


Figure 4-4. The loading state of the asynchronous combobox

Figure 4-5 shows the same combobox once the loader has answered. The results replaced the loading row, and ArrowDown moved the active option onto the first of them.

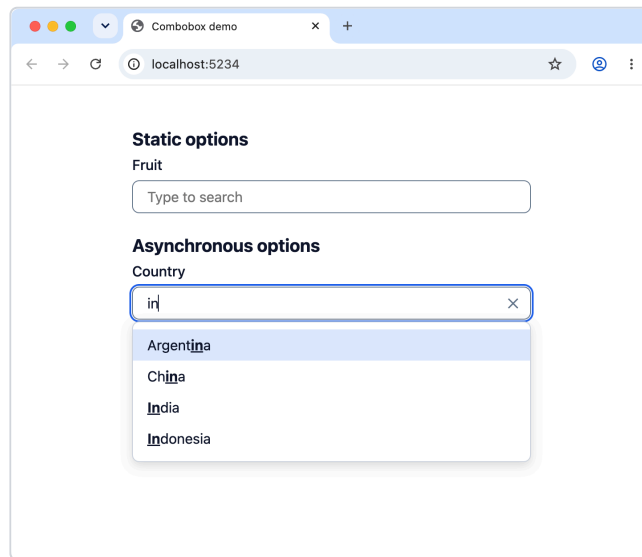


Figure 4-5. Results from the loader, with the first option active

Inspecting the Shadow DOM in Chrome DevTools

Figure 4-6 is the shadow root of the combobox with its list open. The ARIA wiring described in the concepts section is all visible here.



Figure 4-6. The component in Chrome DevTools

Read the numbered callouts from the top:

1. **ds-combobox** is the host, with the attributes the author wrote.
2. **#shadow-root (open)** contains the label, a wrapper, the input, the listbox and a status region.
3. **input** has `role="combobox"`, `aria-expanded="true"`, `aria-controls` pointing at the list, and `aria-activedescendant` naming the highlighted option. Focus never leaves this element.
4. **ul** has `role="listbox"` and the `id` that `aria-controls` refers to. Its options are inside it.

Styling API

Tokens consumed (semantic layer): `--sem-color-surface`, `--sem-color-surface-raised`, `--sem-color-text`, `--sem-color-text-muted`, `--sem-color-text-danger`, `--sem-color-border`, `--sem-color-border-strong`, `--sem-color-action-subtle-bg`, `--sem-color-focus-ring`, `--sem-border-width`, `--sem-focus-ring-width`, `--sem-radius-control`, `--sem-control-padding-block`, `--sem-control-padding-inline`, `--sem-space-stack-sm`, `--sem-shadow-overlay`, `--sem-font-body`, `--sem-font-size-body`, `--sem-font-weight-control`, `--sem-font-weight-strong`.

Component token	Purpose
<code>--ds-combobox-max-height</code>	Maximum height of the popup before it scrolls

Parts: label, input, listbox, option.

Properties: label, name, placeholder, value, disabled, required, options, loadOptions, debounce, empty-text.

Events: ds-select (with detail.option) and change.

Accessibility

Everything in Figure 4-1 is accessibility work, so this section is a checklist of how each piece is verified.

- **Roles and relationships** are asserted in a test: the input is a `combobox`, its `aria-controls` equals the listbox's `id`, and `aria-autocomplete` is `list`.
- **aria-activedescendant** is tested while arrowing through options: it changes with the highlight, wraps around, and keyboard focus stays on the component the whole time.
- **Announcements** are tested by reading the status region: “2 results available”, “Loading”, “No results”, “Could not load options”.
- **An axe audit** runs with the list open.

Warning: The status region announces what a screen reader user needs to hear, but how and when each screen reader reads a live region varies. Automated tests can check that the text is there. They cannot check that it is spoken, so test with a real screen reader before shipping a combobox.

Testing

The asynchronous tests are the most instructive, because they exercise the cases that are hardest to reproduce by hand.

Example 4-6. A stale response must not win

```
it('ignores a slow response that arrives after a newer one', async () => {
  // This loader ignores the abort signal, like a server that cannot cancel work, so the
  // stale answer really does arrive after the fresh one.
  const loader = (q: string) =>
    new Promise<ComboboxOption[]>((resolve) => {
      const delay = q === 'a' ? 150 : 0;
      setTimeout(() => resolve([ { value: q, label: `Result for ${q}` } ]), delay);
    });
  const el = await makeAsync(loader);
  input(el).focus();
  await sendKeys({ type: 'ab' }); // "a" starts a slow request, "ab" a fast one
  await aTimeout(300);
  await settle(el);
  expect(optionEls(el).map((o) => o.textContent!.trim())).to.deep.equal(['Result for
ab']);
});
```

The loader answers the query `a` after 150 milliseconds and the query `ab` immediately, and it deliberately ignores the abort signal, like a server that cannot cancel work in progress. That way the stale answer arrives. The test types both letters, so the slow response to `a` arrives *after* the fast response to `ab`. The expectation is that the list shows only the result for `ab`. With a naive implementation that assigned results as they arrived, the stale answer would overwrite the right one, and the list would show the wrong results for what the user typed.

A second test checks that the previous request is canceled:

Example 4-7. A newer query aborts the older request

```
it('aborts the previous request when the query changes', async () => {
  const aborted: string[] = [];
  const loader = (q: string, signal: AbortSignal) =>
    new Promise<ComboboxOption[]>(() => { signal.addEventListener('abort', () =>
aborted.push(q)); });
  const el = await makeAsync(loader);
  input(el).focus();
  await sendKeys({ type: 'ab' });
  await settle(el);
  expect(aborted).to.include('a');
});
```

The loader never resolves; it only records which queries were aborted. After typing two letters, the first query must appear in that list.

Example 4-8. Output of `npm test`

```
$ npm test

ds-combobox [Chromium]
  ✓ exposes the ARIA combobox structure
  ✓ filters the static options as the user types and opens the list
  ✓ highlights the matching text
  ✓ moves through options with the arrow keys and tracks the active descendant
  ✓ selects the active option with Enter
  ✓ selects an option on click
  ✓ closes on the first Escape and clears on the second
  ✓ restores the selected label when focus leaves with unfinished text
  ✓ shows an empty state
  ✓ clears with the clear button
  ✓ submits the form on Enter when there is nothing to choose, like a native input
  ✓ chooses with Enter instead of submitting while an option is highlighted
  ✓ is accessible with the list open
with an asynchronous loader [Chromium]
  ✓ shows a loading state, then the results
  ✓ shows the loading row even when the previous result was empty
  ✓ ignores a slow response that arrives after a newer one
  ✓ aborts the previous request when the query changes
  ✓ shows an error state when the loader rejects
  ✓ waits for a pause in typing when debounced
in a form [Chromium]
  ✓ submits the selected value and enforces required
  ✓ resets to its initial value
🐛 Browser logs:
  Lit is in dev mode. Not recommended for production! See https://lit.dev/msg/dev-mode for
  more information.
Finished running tests in 2.7s, all tests passed! 🎉
```

Pitfalls

Moving focus into the list. It is tempting to give each option `tabindex="0"` and focus it on arrow keys. That breaks typing: the user can no longer edit the text without first returning focus to the input. Keep focus in the input and use `aria-activedescendant`.

Closing on blur before the click lands. If the popup closes on blur, a click on an option may never arrive, because the press that began the click blurred the input first. Preventing `mousedown`'s default on the list keeps focus where it is.

Keeping an empty list as “the previous results”. The condition that chooses between old results and the loading row must test for *non-empty*, not merely *truthy*. An empty array is *truthy*, so an early search that found nothing made the next search show a blank popup instead of “Loading...”. The bug was found by taking the screenshot for this chapter, which showed a thin empty box where a message should have been, and a test now covers it.

Ignoring the first response. Without cancellation or “latest wins” handling, a response that arrives late overwrites newer results. The Task handles this, but a hand-written `fetch` handler needs its own guard.

Unescaped highlights. Never build the highlighted label by concatenating HTML. Use template parts, so that text stays text.

Empty attributes. Binding `undefined` to an attribute in a Lit template renders an empty attribute, not an absent one. For `aria-activedescendant` that means an attribute that points at nothing. Use `ifDefined`, as the render method does, so that the attribute disappears when there is no value.

Duplicate ids. Every instance needs unique ids for its listbox and options. A counter, as used here, is enough. A hard-coded `id="list"` breaks the second combobox on a page.

Treating the input text as the value. The text and the chosen option are different things. Submitting the text would let a user submit something that is not on the list.

Summary

In this chapter, you built a combobox that follows the ARIA pattern: focus stays in the input, `aria-activedescendant` names the highlighted option, `aria-expanded` and `aria-controls` describe the popup, and a live region announces results. A Lit `Task` loads options and renders loading, empty, error and complete states, and its abort signal and latest-wins behavior make out-of-order responses harmless. Debouncing keeps typing from flooding the server, an `isComposing` guard keeps input methods working, and template-based highlighting keeps data from becoming markup.

In the next chapter, you'll build a date range picker: a calendar grid with two-dimensional keyboard navigation, locale-aware formatting and a small state machine that tracks a range as it is chosen.

Exercises

Each exercise extends this chapter's component. Add a test where the behavior can be tested. Worked solutions, each with a test, are in section E.6 of Appendix E and in `packages/combobox/solutions/`.

1. **Multi-select.** Add a `multiple` attribute. Chosen options become chips inside the field, Backspace on an empty input removes the last chip, and the form submits every chosen value under the same name.
2. **Create new.** When the typed text matches no option, add a final row, `Create "text"`, that selects a new option made from the text. Make sure it is announced.
3. **Fetch with fetch.** Replace the demo's fake server with a real `fetch` call that passes the signal, and handle the `AbortError` so it is not shown as an error.
4. **Grouped options.** Allow options to carry a `group`, render group headings with `role="group"`, and keep arrow navigation moving only through options.
5. **A minimum length.** Add a `min-chars` property so that searching starts only after that many characters, and show a hint such as "Type 2 or more characters" until then.