


```
class LbCard extends HTMLElement {
  connectedCallback() { … }
}
customElements.define(
  "lb-card", LbCard);
```

CORE WEB VITALS

✓ LCP

✓ INP

✓ CLS

BUILD REAL WEB APPS

# Modern JavaScript for the Web

## Fast, accessible web apps with vanilla JavaScript and the web platform

DOM · Web Components · routing · IndexedDB · service workers

DOM · STATE · FETCH · OFFLINE · PERFORMANCE · SECURITY

THENEXBOOKS

# Modern JavaScript for the Web

*Build Fast, Accessible Web Apps with Vanilla JavaScript and the Web Platform*

*DOM · Components · Routing · Offline · Performance · Security*

First Edition

THENEXBOOKS

**Modern JavaScript for the Web**
Build Fast, Accessible Web Apps with Vanilla JavaScript and the Web Platform

First Edition, 2026

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

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

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Browsers and web APIs evolve continuously; always check current browser support and the documentation for the features you use.

JavaScript is a trademark of Oracle Corporation in the United States. Chrome is a trademark of Google LLC; Firefox is a trademark of the Mozilla Foundation; Safari is a trademark of Apple Inc.; Edge is a trademark of Microsoft Corporation. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is not affiliated with, sponsored by, or endorsed by any of these companies.

Loomboard, the running example in this book, is fictional, as are all teams, people, and organizations mentioned in its examples. Any resemblance to real companies or products is coincidental.

# Preface

The web platform has changed more in the last decade than most developers noticed. Browsers now ship native modules, a capable networking stack with cancellation, a real database in IndexedDB, component primitives with custom elements and shadow DOM, accessible dialogs and popovers, observers for visibility and size, animation APIs, service workers for offline apps, and a rich internationalization library — all without installing anything. Many of the problems that once required a large framework now have a standard answer built into every modern browser.

Frameworks remain excellent tools, and many teams should use one. But every framework is built on the same platform, and developers who understand that platform write better applications with or without a framework: they know why an update is slow, why a form is inaccessible, why a token leaked, why the app broke offline, and what the framework is actually doing for them.

This book teaches you to build complete, production-quality web applications with modern JavaScript and the web platform itself. We start from how browsers load and run code, work through the DOM, events, forms, and accessibility, design an application architecture with state, routing, and components, connect it to a server with robust networking, real-time updates, and offline storage, and then make it fast, secure, installable, tested, and deployed.

## Who This Book Is For

- **Front-end developers** who want to understand the platform beneath their framework and build lighter, faster apps.
- **Back-end and full-stack developers** who need to build solid browser UIs without adopting a heavy stack.
- **Developers who learned JavaScript basics** and want to go from scripts on a page to real applications.
- **Tech leads** choosing how much framework a project actually needs.

You should already know basic JavaScript (variables, functions, arrays, objects, `async` / `await`) and some HTML and CSS. Chapter 2 is a compact refresher of the modern syntax we use throughout.

## What You Will Learn

- How browsers parse, load, and run your code, and how to set up a modern project with Vite, ES modules, and linting.
- Efficient DOM updates, event delegation, pointer and keyboard input, forms with native validation, and safe templating.
- Accessibility with JavaScript: focus management, ARIA, dialogs, and keyboard interaction.
- Application architecture without a framework: modules, components, a small reactive store, and a History API router.

- Web Components with custom elements, shadow DOM, and slots; styling and motion from JavaScript.
- A robust API client with timeouts, retries, and cancellation; IndexedDB and the Cache API; Server-Sent Events and WebSockets.
- Authentication in the browser, file uploads, drag and drop, and images.
- Core Web Vitals, loading and runtime performance, Web Workers, and front-end security including CSP.
- Progressive Web Apps with service workers and offline sync, internationalization with Intl, testing with Vitest and Playwright, and deployment.

## How This Book Is Organized

**Part I — Foundations for Modern Web Development** (Chapters 1–4) surveys the platform, refreshes modern syntax, sets up the project, and explains how browsers run code. **Part II — DOM, Events, and Accessible UI** (Chapters 5–9) builds the interface layer. **Part III — Architecture of a Front-End App** (Chapters 10–14) organizes it into modules, state, routes, and components. **Part IV — Data and Networking** (Chapters 15–19) connects it to the server, the network, and local storage. **Part V — Performance, Security, and Progressive Web Apps** (Chapters 20–24) makes it fast, safe, installable, and ready for users around the world. **Part VI — Testing, Shipping, and Case Study** (Chapters 25–28) tests, monitors, deploys, and assembles the whole application. The appendices provide a cheat sheet, a web API reference with templates, a production checklist, and a glossary.

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

## The Running Example: Loomboard

Throughout the book we build **Loomboard**, a fictional kanban-style task board for small teams. Our sample team is Orchard Street Makers, a community makerspace planning its Spring Workshop Series: cards move from "To do" to "Done", members are assigned, due dates approach, teammates edit the same board at the same time, and volunteers keep working on a patchy connection. That gives us every classic front-end challenge — accessible drag and drop, forms and validation, optimistic updates, live collaboration, offline sync, image uploads, and a fast first load — in one realistic app you can build and run.

## Browsers and Conventions

Examples target the current versions of the major browsers and use only standard web APIs. Where an API is new or not yet supported everywhere, we say so, show how to detect it, and provide a fallback; always check current support on MDN or a similar reference before relying on a newer

feature. Examples use modern JavaScript (ES2022 and later), plain CSS with custom properties, Vite for development and builds, Vitest for unit tests, and Playwright for browser tests.

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

> **NOTE**
>
> Background information, browser support details, or a useful aside.

> **WARNING**
>
> A common mistake that causes bugs, accessibility barriers, security holes, or slow pages.

Code appears in `monospace`. Long lines are wrapped to fit the page, and file names above code blocks follow the Loomboard project layout from Chapter 3.

# Contents

Preface

PART I — FOUNDATIONS FOR MODERN WEB DEVELOPMENT

1 The Modern Web Platform: What Browsers Give You Today

2 Modern JavaScript Essentials for Web Developers

3 Project Setup: Vite, ES Modules, Linting, and Formatting

4 How Browsers Load and Run Your Code: Parsing, Scripts, Modules, and the Event Loop

PART II — DOM, EVENTS, AND ACCESSIBLE UI

5 The DOM in Depth: Selecting, Creating, and Updating Elements Efficiently

6 Events: Delegation, Custom Events, Pointer, and Keyboard Input

7 Forms: Constraint Validation, FormData, and Accessible Error Messages

8 Rendering Lists and Templates Safely: template, DocumentFragment, and Avoiding XSS

9 Accessibility with JavaScript: Focus Management, ARIA, Dialogs, and Keyboard Interaction

PART III — ARCHITECTURE OF A FRONT-END APP

10 Structuring a Vanilla JavaScript App: Modules, Components, and Separation of Concerns

11 State Management Without a Framework: Stores, Reactivity, and Immutable Updates

12 Client-Side Routing with the History API

13 Web Components: Custom Elements, Shadow DOM, Slots, and Templates

14 Styling and Motion from JavaScript: Classes, CSS Custom Properties, Web Animations, and View Transitions

PART IV — DATA AND NETWORKING

15 Fetch in Practice: An API Client with Errors, Timeouts, Retries, and AbortController

16 Storing Data in the Browser: Web Storage, IndexedDB, and the Cache API

17 Real-Time Updates: Server-Sent Events, WebSockets, and Polling

18 Authentication in the Browser: Sessions, Cookies, Tokens, and Protected Routes

19 Files, Images, and Drag and Drop: File API, Uploads, Canvas, and Pointer Events

**PART V — PERFORMANCE, SECURITY, AND PROGRESSIVE WEB APPS**

20 Loading Performance: Core Web Vitals, Resource Hints, and Code Splitting

21 Runtime Performance: Rendering, Layout Thrashing, Long Tasks, and Web Workers

22 Front-End Security: XSS, Content Security Policy, CSRF, CORS, and Dependencies

23 Progressive Web Apps: Service Workers, Offline Sync, and Installability

24 Internationalization and Localization with Intl

**PART VI — TESTING, SHIPPING, AND CASE STUDY**

25 Testing Front-End Code: Unit Tests with Vitest and End-to-End Tests with Playwright

26 Debugging and Monitoring: DevTools, Error Reporting, and Real User Monitoring

27 Building and Deploying: Production Builds, Environment Configuration, Caching, and Static Hosting

28 Case Study: Building Loomboard from Empty Folder to Production

**APPENDICES**

A Modern Web JavaScript Cheat Sheet

B Web API Reference and Templates

C Production Checklist

D Glossary

PART I

# Foundations for Modern Web Development

What the modern web platform gives you out of the box, the JavaScript syntax we rely on, a clean Vite project for Loomboard, and how browsers actually parse, load, and run your code.

1 · The Modern Web Platform
2 · Modern JavaScript Essentials
3 · Project Setup with Vite
4 · How Browsers Run Your Code

CHAPTER 1 · PART I — FOUNDATIONS FOR MODERN WEB DEVELOPMENT

# The Modern Web Platform: What Browsers Give You Today

*Modern browsers ship a remarkably complete application platform: native dialogs, form validation, component models, offline storage, background threads, and a world-class internationalization library. This chapter maps that platform, introduces Loomboard — the task board we build throughout the book — and sets out the habits that let you use new features safely.*

---

## Why Build on the Platform?

For a long stretch of the web's history, building an application in the browser meant fighting the browser. Each engine exposed slightly different DOM methods, event models diverged, and basic interface pieces such as modal dialogs, date inputs, or reliable layout primitives either did not exist or behaved inconsistently. Libraries and frameworks grew up to paper over those gaps, and they did an enormous amount of good. Many developers learned the web through those abstractions and never had a reason to look underneath.

The situation today is very different. The major browser engines now implement a shared set of standards far more consistently, and they update themselves automatically every few weeks. Features that once required a library — querying elements with CSS selectors, making network requests with promises, cloning complex data, observing when an element scrolls into view, showing an accessible modal dialog — are built in. The platform has quietly become an application framework of its own, with a very long support horizon: code written against web standards tends to keep working for many years, because browsers go to great lengths not to break existing pages.

This book is about building real, production-quality applications with that platform directly: the DOM, events, forms, accessibility, state, routing, Web Components, networking, offline storage, performance, security, Progressive Web App features, testing, and deployment. We write *vanilla* JavaScript — no UI framework — but we use modern tooling (Vite, ESLint, Prettier, Vitest, Playwright) because good tooling is not the same thing as a framework. You will leave with two kinds of value:

- **The ability to ship without a framework** when that is the right call: small and medium applications, embeddable widgets, internal tools, content sites with islands of interactivity, and anything where bundle size and longevity matter.
- **A clear mental model of what frameworks do under the hood.** When you understand how a store notifies subscribers, how a router intercepts navigation, or how a component re-renders a list safely, you debug framework applications faster and choose between frameworks

with more confidence.

We assume you already know JavaScript basics and some HTML and CSS. We do not re-teach the language in depth; Chapter 2 is a compact refresher focused on the features web applications use every day, and elsewhere we use modern syntax freely. What we *do* explain at length is the browser: how it loads your code, how it renders, how it dispatches events, and how its APIs fit together.

## Meet Loomboard

Abstract advice about "building applications" is easy to agree with and hard to apply. So instead of isolated snippets, every chapter contributes to one application: **Loomboard**, a kanban-style task board for small teams. It is fictional, but it is shaped like the software many of us build for a living, and it exercises almost every part of the platform.

In Loomboard, users sign in and open a **board**. A board has four **columns** — "To do", "In progress", "Review", and "Done" — and each column holds **cards**. Users create and edit cards, drag them between columns, assign members, add due dates and labels, leave comments, and attach images. When a teammate changes something, everyone else sees the update live. When the network drops, users keep working, and their changes synchronize when the connection returns.

Our sample data belongs to a fictional community makerspace, the team "Orchard Street Makers". Their board, `board_7`, is called "Spring Workshop Series". The team has four members: Ana Lima (`u_1`, the board owner), Ben Carter (`u_2`), Chloe Nakamura (`u_3`), and Dev Patel (`u_4`). You will meet the same four cards again and again:

| Card | Title | Column | Labels | Assignee |
|---|---|---|---|---|
| `card_101` | Book the laser cutter training | To do | logistics | Ben Carter (due 2026-04-03) |
| `card_102` | Design the workshop poster | In progress | design | Chloe Nakamura |
| `card_103` | Order plywood sheets | Review | purchasing | Dev Patel |
| `card_104` | Email volunteers | Done | comms | Ana Lima |

Throughout the book, the data is made of plain JavaScript objects with stable names. Here is the shape of the model, written as JSDoc type definitions so that editors can offer autocompletion without any TypeScript compiler:

src/types.js (reference only)

```
/**
 * @typedef {Object} Board
 * @property {string} id            e.g. "board_7"
 * @property {string} name          e.g. "Spring Workshop Series"
 * @property {Column[]} columns
 *
 * @typedef {Object} Column
 * @property {"col_todo"|"col_doing"|"col_review"|"col_done"} id
```

```
 * @property {string} title
 * @property {string[]} cardIds     ordered card ids
 *
 * @typedef {Object} Card
 * @property {string} id
 * @property {string} title
 * @property {string} description
 * @property {string} columnId
 * @property {string[]} labels
 * @property {string|null} assigneeId
 * @property {string|null} dueDate  ISO date, e.g. "2026-04-03"
 * @property {number} position
 * @property {string} updatedAt     ISO timestamp
 * @property {number} version       incremented on every server update
 *
 * @typedef {Object} Member
 * @property {string} id
 * @property {string} name
 * @property {string} avatarUrl
 *
 * @typedef {Object} Comment
 * @property {string} id
 * @property {string} cardId
 * @property {string} authorId
 * @property {string} body
 * @property {string} createdAt
 */
```

Loomboard talks to a REST-style JSON API under `/api`: it loads a board with `GET /api/boards/:boardId` and its cards with `GET /api/boards/:boardId/cards`, creates cards with `POST /api/cards`, and updates them with `PATCH /api/cards/:id`, sending the card's version in an `If-Match` header so the server can reject stale edits. Live updates arrive as Server-Sent Events from `GET /api/boards/:boardId/events`. The real backend is out of scope — this is a front-end book — but in Chapter 3 we build a tiny mock API inside the Vite dev server so that everything runs on your machine.

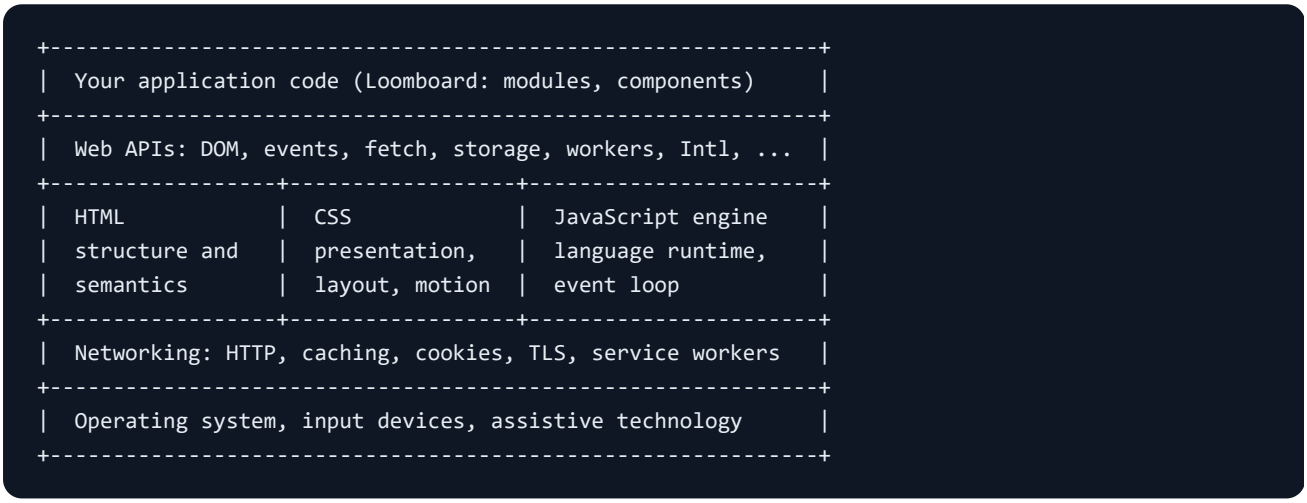
The table below shows how Loomboard's features map to the platform and to the chapters where we build them. Treat it as a map of the book as much as a map of the application.

| Loomboard feature | Platform pieces | Chapter |
|---|---|---|
| Render board, columns, cards | DOM, `<template>`, DocumentFragment | 5, 8 |
| Click, keyboard, drag | Events, delegation, Pointer Events | 6, 19 |
| Edit card form | Constraint validation, FormData | 7 |
| Card dialog | `<dialog>`, focus, ARIA | 9 |
| App structure and state | ES modules, stores, subscriptions | 10, 11 |
| Deep links to cards | History API | 12 |

| | | |
|---|---|---|
| Reusable widgets | Custom Elements, Shadow DOM | 13 |
| Smooth card moves | Web Animations, View Transitions | 14 |
| Talking to the server | fetch, AbortController | 15 |
| Offline edits | IndexedDB, Cache API, service worker | 16, 23 |
| Live updates | Server-Sent Events, WebSockets | 17 |
| Sign-in | Cookies, sessions, protected routes | 18 |
| Dates and counts | `Intl` formatters | 24 |

## The Platform in Layers

It helps to picture the browser as a stack of layers, each with a distinct job. When something goes wrong in production, the first diagnostic question is usually "which layer is this?" — a markup problem, a styling problem, a script problem, or a network problem.




Two points in that diagram are easy to miss. First, JavaScript the *language* and the Web APIs are different things. The language (standardized by Ecma's TC39 committee as ECMAScript) gives you objects, functions, promises, modules, and classes; it knows nothing about documents or networks. The Web APIs — `document`, `fetch`, `localStorage`, `customElements` — are provided by the browser, mostly specified by the WHATWG and the W3C. The same language runs in Node.js with a completely different set of host APIs, which is why a function like `structuredClone` can exist in both environments while `document` exists in only one.

Second, assistive technology sits at the bottom of the stack, next to the keyboard and mouse. Screen readers do not read your pixels; they read the accessibility tree the browser builds from your HTML and ARIA attributes. When you choose the right element, you are programming that layer too. That is one reason this book leans so heavily on native HTML: a real `<button>` is focusable, activates on Enter and Space, and announces itself as a button, all for free.

## What HTML Gives You Now

HTML has grown a set of interactive elements that replace a surprising amount of custom JavaScript. Many teams still hand-build these widgets out of `<div>` elements out of habit, and in doing so they lose keyboard support, focus handling, and screen-reader semantics. Before writing a widget, check whether the platform already has one.

### Dialogs, disclosure, and popovers

The `<dialog>` element with its `showModal()` method is supported in all major browsers in recent versions. A modal dialog opened this way renders in the *top layer* above everything else, makes the rest of the page inert, moves focus inside, and closes on the Escape key. The `<details>` and `<summary>` pair gives you an accessible disclosure widget with no script at all. The Popover API (Baseline 2024) turns any element into a light-dismiss popover with a single attribute — ideal for menus and tooltips-with-content.

Here is a first sketch of Loomboard's card details, using nothing but HTML. It already works with the keyboard and is announced correctly by screen readers:

sketches/card-dialog.html

```
<button type="button" id="open-card">Book the laser cutter training</button>

<dialog id="card-dialog" aria-labelledby="card-dialog-title">
  <form method="dialog">
    <h2 id="card-dialog-title">Book the laser cutter training</h2>
    <p>Due <time datetime="2026-04-03">April 3, 2026</time></p>

    <details>
      <summary>Activity (2 comments)</summary>
      <p>Comments load here.</p>
    </details>

    <button type="button" popovertarget="card-menu">More actions</button>
    <div id="card-menu" popover>
      <button type="button">Duplicate</button>
      <button type="button">Archive</button>
    </div>

    <button value="close">Close</button>
  </form>
</dialog>
```

The only JavaScript needed is one line to open it: `document.querySelector("#card-dialog").showModal()`. The `<form method="dialog">` closes the dialog when its submit button is pressed and sets `dialog.returnValue` to that button's value. We build the complete, accessible card dialog — with focus restoration and form handling — in Chapter 9.

### Forms that validate themselves

Attributes such as `required`, `maxlength`, `min`, `pattern`, and `type="email"` give you built-in validation, and the Constraint Validation API (`checkValidity()`, `setCustomValidity()`, the `validity` object) lets you customize the messages and timing from JavaScript. The `FormData` object reads an entire form in one call. Chapter 7 builds Loomboard's card editor on these pieces, including accessible error messages.

### Templates and inert content

The `<template>` element holds markup that is parsed but not rendered, ready to be cloned. It is the safe foundation for rendering lists without string concatenation, which we use in Chapter 8. The `inert` attribute, supported in all major browsers in recent versions, removes a subtree from focus order and the accessibility tree at once — perfect for a side panel that is visually hidden but still in the DOM.

> **TIP**
> Before reaching for a library widget, search MDN for the native element first: "dialog", "details", "popover", "datalist", "input type=date". Even when the native element is not quite enough, it is usually the best base to enhance.

## What CSS Gives You Now

A JavaScript book spends time on CSS because a large share of the code in older applications existed only to do things CSS now does better. Every time you move work from a script into a stylesheet, the browser can optimize it, it keeps working if a script fails, and there is less code for you to maintain.

- **Custom properties** (`--lb-accent`) are live variables that cascade and can be changed from JavaScript with one call to `style.setProperty()`. Loomboard's theming — including a dark mode and a per-board accent color — is built on them.
- **Grid and Flexbox** handle the board layout: four columns that scroll horizontally on small screens and share space on large ones, without measuring anything in script.
- **The `:has()` selector** lets a parent react to its children. A column can highlight itself when it contains a card being dragged, with `.lb-column:has(.is-dragging)`, rather than toggling classes by hand.
- **Container queries** let a component adapt to the space *it* is given rather than to the viewport. A card can show a compact layout in a narrow column and a richer one in a wide dialog.
- **Transitions, animations, and `prefers-reduced-motion`** cover most interface motion declaratively; Chapter 14 adds the Web Animations API for motion that must be driven from script.

src/styles/tokens.css (preview)

```
:root {
  --lb-accent: #3b5bdb;
  --lb-surface: #ffffff;
  --lb-surface-muted: #f1f3f5;
  --lb-text: #1f2328;
  --lb-radius: 8px;
  --lb-space: 0.75rem;
}

@media (prefers-color-scheme: dark) {
  :root {
    --lb-surface: #1b1e23;
    --lb-surface-muted: #262a31;
    --lb-text: #e9ecef;
  }
}

.lb-column:has(.is-dragging) {
  outline: 2px dashed var(--lb-accent);
}

.lb-board {
  display: grid;
  grid-auto-flow: column;
  grid-auto-columns: minmax(16rem, 1fr);
  gap: var(--lb-space);
  overflow-x: auto;
}
```

Notice how little of that would need JavaScript. The script's job is to put the right classes and data on the right elements; the stylesheet decides how they look. Keeping that division clean is one of the architectural themes of Part III.

## JavaScript and the Web APIs

On the scripting side, the list of built-in capabilities is long. These are the APIs Loomboard relies on, all supported in all major browsers in recent versions:

| Need | Built-in API | Replaces |
|---|---|---|
| Load code in pieces | ES modules, dynamic `import()` | Script concatenation, AMD loaders |
| Find elements | `querySelector`, `closest` | Selector libraries |
| Network requests | `fetch`, `AbortController` | XHR wrappers |
| Deep-copy data | `structuredClone` | JSON round trips, clone helpers |
| Lazy work on scroll | `IntersectionObserver` | Scroll listeners with math |
| React to size changes | `ResizeObserver` | Window resize polling |

| | | |
|---|---|---|
| Reusable components | Custom Elements, Shadow DOM | Widget frameworks |
| Dates, numbers, plurals | `Intl.*` | Large formatting libraries |
| Structured local data | IndexedDB | Ad hoc storage plugins |
| Offline and caching | Service Workers, Cache API | AppCache (removed) |
| Heavy computation | Web Workers | Blocking the main thread |

A small example shows the flavor. The following function loads Loomboard's cards and gives up after five seconds, using only built-in APIs. We build the real client, with error types and retries, in Chapter 15; this is only a taste.

sketches/load-cards.js

```
export async function loadCards(boardId) {
  const response = await fetch(`/api/boards/${encodeURIComponent(boardId)}/cards`, {
    headers: { Accept: "application/json" },
    signal: AbortSignal.timeout(5000),
  });
  if (!response.ok) {
    throw new Error(`Loading cards failed with HTTP ${response.status}`);
  }
  return response.json();
}

const cards = await loadCards("board_7");
const formatter = new Intl.DateTimeFormat("en-US", { dateStyle: "medium" });
for (const card of cards) {
  const due = card.dueDate ? formatter.format(new Date(card.dueDate)) : "no due date";
  console.log(`${card.title}: ${due}`);
}
```

Every line of that is standard: template literals, `async` / `await`, top-level `await` in a module, `fetch`, `AbortSignal.timeout()`, and `Intl.DateTimeFormat`. Ten years ago the same code would have required a promise polyfill, an HTTP library, and a date library.

> **NOTE**
>
> An ISO date string such as `"2026-04-03"` is parsed by `new Date()` as midnight *UTC*, so formatting it in a time zone west of UTC can show the previous day. Loomboard's formatting helpers in `src/lib/format.js` handle date-only values carefully; Chapter 24 explains the details.

## Evergreen Browsers, Standards, and Baseline

Modern desktop and mobile browsers are *evergreen*: they update automatically and frequently. In practice this means the question "which browser versions do we support?" has been replaced by a better one: "which *features* can we rely on for our users?" The answer depends on your audience. A

tool used inside a company that manages its laptops can adopt features faster than a public site that must work on older phones that no longer receive updates.

Web features move through a fairly predictable path. They are discussed and specified in standards bodies — the WHATWG maintains the HTML and DOM living standards, the W3C's working groups handle CSS and many APIs, and TC39 evolves the JavaScript language through numbered proposal stages. Browser engines implement features, sometimes one at a time, sometimes behind flags first. Only when all major engines ship a feature can you rely on it without a fallback.

**Baseline** is a shared label, shown on MDN and elsewhere, that summarizes this. A feature is "Baseline newly available" when it works in the current versions of all major browsers, and "Baseline widely available" once it has been in them long enough that most users' browsers support it. When this book says a feature is "supported in all major browsers in recent versions", we mean it has reached Baseline. When support is uneven or recent — the Navigation API, cross-document View Transitions, `scheduler.yield()`, `URLPattern`, Background Sync — we say so, and we always pair the feature with detection and a fallback.

> **WARNING**
>
> Do not trust any book, blog post, or AI answer — including this one — for the exact current support status of a new API. Support changes every few weeks. Check MDN's compatibility tables or the Baseline status before relying on a feature, and re-check when you upgrade your support targets.

## Feature Detection and Progressive Enhancement

Because features arrive at different times, well-built web applications *detect* capabilities instead of guessing them from the browser's name. User-agent sniffing is fragile: strings lie, change, and get frozen. Feature detection asks the only question that matters — "does this API exist here?" — and it keeps working as browsers improve.

src/lib/support.js

```
export const supports = {
  viewTransitions: typeof document.startViewTransition === "function",
  navigationApi: "navigation" in window,
  popover: typeof HTMLElement.prototype.togglePopover === "function",
  serviceWorker: "serviceWorker" in navigator,
  schedulerYield: typeof globalThis.scheduler?.yield === "function",
  cssHas: CSS.supports("selector(:has(*))"),
  containerQueries: CSS.supports("container-type: inline-size"),
};

/** Run an update with a view transition when available, instantly otherwise. */
export function withTransition(update) {
  if (!supports.viewTransitions) {
    update();
    return Promise.resolve();
```

```
  }
  return document.startViewTransition(update).finished;
}
```

The pattern in `withTransition` is the one we use for every newer API: the enhanced path and the fallback produce the same final state; only the polish differs. A user on an older browser sees the card jump to its new column instead of animating there. Nothing breaks.

That idea generalizes into **progressive enhancement**: start from a baseline that works with the most widely supported features, then layer improvements on top when the browser can handle them. For Loomboard this means, for example, that the "New card" form is a real `<form>` with a real submit button, so it is keyboard-accessible and meaningful even before our scripts attach behavior; that cards are movable with a keyboard-accessible "Move to…" menu as well as with drag and drop; and that offline support is an enhancement layered on top of a working online app, not a prerequisite for it.

In CSS, the equivalent tool is the `@supports` rule, and it is often cleaner than detecting in script:

```
.lb-card { padding: var(--lb-space); }

@supports (container-type: inline-size) {
  .lb-column { container-type: inline-size; }
  @container (min-width: 22rem) {
    .lb-card { display: grid; grid-template-columns: 1fr auto; }
  }
}
```

## Accessibility and Performance Are Platform Features

Two qualities separate a professional web application from a demo: it is usable by everyone, and it feels fast. Both are easier to achieve with the platform than against it.

### Accessibility

Browsers expose an accessibility tree derived from your markup. Native elements come with correct roles, states, and keyboard behavior; custom widgets must recreate all of that with ARIA attributes and event handling, and they usually get something wrong. The first rule of ARIA is often summarized as "do not use ARIA if a native element will do". In Loomboard we use `<button>` for actions, `<a href>` for navigation, `<dialog>` for modal content, `<form>` for input, and headings that describe the page's structure. We add ARIA only where HTML has no equivalent — for example, a live region that announces "Card moved to Review" after a drag, which Chapter 9 builds.

### Performance

Google's Core Web Vitals give teams a shared vocabulary for user-perceived performance. The published "good" thresholds, measured at the 75th percentile of page loads, are:

| Metric | What it measures | Good |
|---|---|---|
| Largest Contentful Paint (LCP) | How quickly the main content appears | ≤ 2.5 s |
| Interaction to Next Paint (INP) | How quickly the page responds to input | ≤ 200 ms |
| Cumulative Layout Shift (CLS) | How much content jumps around | ≤ 0.1 |

INP replaced First Input Delay (FID) as a Core Web Vital in March 2024. It is especially relevant to applications like Loomboard, where users click, type, and drag constantly: every interaction that blocks the main thread for too long counts against it. The platform helps here too. Modules load in parallel, the browser's preload scanner discovers resources early, the Cache API and HTTP caching avoid repeat downloads, and Web Workers move heavy computation off the main thread. Chapter 4 explains the loading pipeline and the event loop that underlie all of this, and Chapters 20 and 21 turn it into practice. Throughout, our rule is simple: describe the mechanism, then *measure* — never assume a change is faster because it sounds faster.

## A Security Baseline from Day One

A browser application runs code from many sources on a page that also holds a user's session. Most front-end security incidents come from a small number of mistakes, and it is far cheaper to avoid them from the first commit than to retrofit a fix.

- **Never put untrusted data into HTML with `innerHTML`.** A card title such as `<img src=x onerror=alert(1)>` becomes executable code. Use `textContent`, DOM construction, or a template, as Chapter 8 shows.
- **Keep sessions in HttpOnly, Secure, SameSite cookies**, not in `localStorage`, where any injected script can read them. Loomboard's `POST /api/session` sets such a cookie; Chapter 18 covers the trade-offs.
- **Ship a Content Security Policy** that restricts where scripts may come from, preferably with nonces or hashes (Chapter 22).
- **Treat dependencies as code you run.** Every package in your build can execute during installation or end up in your bundle.

## When a Framework Is Still the Right Choice

This is not an anti-framework book. Frameworks encode years of hard-won solutions and offer large ecosystems, hiring pools, and conventions that help big teams move in step. If your application has hundreds of screens, many teams contributing to one codebase, or heavy server-rendering requirements, a mature framework may be the more economical choice, and everything in this book will still make you better at using it.