pb powerbunch



SAMPLE

The Blueprint

Building a Component Library

Architecture, design tokens, accessibility, frameworks and operations for your own component library, with Mosaik as a case study

16 chapters | Code examples in Lit 3 | Checklists and glossary | Mosaik case study

Original pages from the 271-page handbook: contents, chapter 01 and excerpts from chapters 02 and 04. The remaining pages are omitted, and the page numbers refer to the entire handbook.

Version 1.0 · As of November 15, 2026 · powerbunch GmbH · Case study: Mosaik by André Wehlert

Contents

How to read the boxes

IN PRACTICE
Experience and proven approaches.

PITFALL
Typical mistakes and their consequences.

HOW MOSAIK DOES IT
Documented solutions from the case study.

01	**How to use this handbook**	**7**
	What a component library is	7
	Who this handbook is for	9
	Three reading paths through the book	10
	The reference project	11
	The Mosaik case study	13
	Conventions in this book	14
02	**Build your own library, adapt, or buy?**	**17**
	Asking the right question	17
	Five decision criteria	18
	The realistic options	19
	Total cost over five years: an estimate	23
	Business case and risks	25
	Exit strategy: plan the exit before the entry	26
03	**Architecture and fundamental decisions**	**29**
	Why web components are the common base	29
	Lit 3 as the base and the alternatives	30
	Shadow DOM, light DOM or both	32
	Repository, packages and dependencies	33
	Defining the browser baseline	35
	Names, prefixes and the custom element registry	36
	TypeScript and decorators	38
	Build, modules and tree shaking	40
04	**The component model**	**43**
	The public API as a contract	43
	Attributes and properties	44
	Events	46
	Slots, sub-elements and parts	47
	Reusable behavior with controllers and mixins	49
	State and form participation	51
	Focus and lifecycle	55

Performance and definition of done 56

05 **Design tokens and theming** 58

What tokens are and what they are not 58

Three levels: base, semantic, component 59

The DTCG format 2025.10 60

The token pipeline from JSON to CSS, TypeScript and Tailwind 63

Color ramps, OKLCH and contrast 66

Themes and color schemes at runtime 68

Component tokens, density, shape and typography 70

Alignment with Figma 71

06 **Styling and customization** 74

Encapsulation through shadow DOM and its consequences 74

The public styling API as a contract 75

Variants, sizes and density 78

Light DOM, Tailwind and global styles 80

Constructable stylesheets and performance 81

Focus, high contrast and writing direction 83

Versioning the styling API 84

Typical mistakes and checklist 86

07 **Accessibility** 88

Why the library is the most effective place 88

Patterns from the ARIA Authoring Practices Guide 89

Shadow DOM: pitfalls and solutions 91

Keyboard and focus 94

Focus management in overlays 96

Forms: labels, errors, validation 98

Motion, contrast and forced colors 102

Testing and documenting 103

08 **One element for all frameworks** 107

One source, many thin adapters 107

The Custom Elements Manifest as a contract 108

Architecture of a wrapper generator 110

React: native support, still a wrapper 113

Angular: standalone components, signal inputs and forms 115

Vue 3 and Svelte 5 120

Type definitions and server-side rendering 123

Testing and accepting wrappers 125

09 The foundation: services beyond components 127

What belongs in the library and what in the application 127

Overlays, portals and positioning 129

Internationalization 132

Dependency injection and router 135

Accessible drag and drop 137

Layout and motion: breakpoints, container queries, animations 138

Icons: registry, loading on demand and licenses 140

Decision aid: does it belong in the library? 143

10 Quality: tests and checks 145

Test strategy for a library 145

Component tests in a real browser 146

Interaction, accessibility and cross-browser 149

Visual regression 151

API stability and types 153

Wrappers, packages and budgets 154

The CI pipeline 156

Definition of done and typical mistakes 157

11 Documentation for humans and machines 160

Documentation types and their readers 160

The API reference from the manifest 161

Examples, playground and documentation platform 163

Writing guidelines: descriptions with meaning 165

Documentation for AI assistants 167

An MCP server for the design system 169

Quality assurance for the AI documentation 172

12 Tools around the library 175

Which tools pay off when 175

Playground per component 176

Themes: editor, generator, palettes, contrast and audit 179

Token bridge, motion and typography 181

Visual designer, recipes and starter templates 182

Icon gallery and AI context generator 185

Order and dogfooding 186

13 Publish, version, distribute 189

Semantic Versioning for component libraries 189

Changesets: versions and changelogs in the monorepo 191

The package format: ES modules, exports, types, manifest 193

Choosing the registry 195
CDN delivery and import maps 196
Deprecation, migration guides and codemods 197
Release notes, pipeline and proof of origin 199

14 Operations and governance 203

Team models and roles 203
Contributions and decisions 205
Roadmap, prioritization and support 207
Measuring adoption without inventing numbers 209
Communication: changelog, release notes and office hours 211
Funding and internal cost allocation 212
Lifecycle of components 213

15 Roadmap: your own library in 12 months 215

The roadmap at a glance 215
Remediation of an existing library 216
Phase 1, months 1 to 2: foundation 218
Phase 2, months 3 to 6: core components 220
Phase 3, months 7 to 9: expansion and frameworks 222
Phase 4, months 10 to 12: maturity and operations 223
The first 25 components in a justified order 224
Team setup and smaller teams 226

16 Case study: Mosaik 229

Profile 229
Origins: four eras and one lesson 231
The key decisions 232
What other teams can adopt and what not 234
Open points from a buyer’s perspective 238
Lessons 239

A Checklists 241

Ready for your own library 241
Architecture decisions 241
New component: definition of done 242
Design tokens 243
Accessibility per component 243
Framework wrappers 244
Release 244
Deprecation 245
Documentation 245

Governance 246

B Glossary 247

C Sources 253

D License terms and publisher 264

CHAPTER 01

How to use this handbook

This chapter defines the key terms, shows three reading paths through the book and introduces the reference project and the Mosaik case study. Afterwards you know which chapters to read first for your role and how to interpret code, boxes and estimates.

01 What a component library is
02 Who this handbook is for
03 Three reading paths through the book
04 The reference project
05 The Mosaik case study
06 Conventions in this book

What a component library is

A component library is a versioned collection of reusable user interface building blocks that is delivered as code. Every building block has a documented public interface made of attributes, properties, events, slots and styling hooks. Product teams install the library as a package and build their interfaces with it. The word “versioned” is decisive here: A library is a product with releases, a changelog and stability commitments, not a folder of snippets to copy.

The Nielsen Norman Group describes a component library as a collection of predefined, reusable UI elements. Each entry typically includes a name, a description, attributes, states and code. This handbook sharpens the definition for practical use with three characteristics that separate a real library from a collection of good intentions.

- **Delivered as a package.** Teams install a specific version and upgrade deliberately. Copying source code and adapting it locally does not consume the library. It starts a fork.
- **Behavior included.** Focus management, keyboard operation, states such as “disabled” or “loading” and accessibility belong to the component. An application should not have to retrofit them.
- **Public interface as a contract.** What is documented counts as a commitment: From version 1.0 on, it stays compatible within a major version according to Semantic Versioning. For versions below 1.0, the explicitly published commitments apply. What is not documented, such as internal classes or the structure inside the shadow DOM, may change at any time.

How it differs from related terms

In projects, five terms are often mixed up. The confusion is expensive, because each term implies a different budget, a different team and different success criteria. The following table defines how this handbook uses the terms. It is not a standard but a working definition that you can adopt in your company or deliberately define differently.

Term	What it is	What it delivers	Typical gap
Design system	The complete set of design principles, rules, design tokens, components, patterns, documentation and the team that maintains all of it	Consistency across products and a shared language between design and development	Without a component library it remains a set of rules that every team implements on its own
Component library	Versioned code building blocks delivered as a package, with a documented API	Working building blocks including behavior, states and accessibility	Without a design system, the rules for when to use which component are missing

Term	What it is	What it delivers	Typical gap
UI kit	A collection of designed elements, usually as a design file (for example a Figma library) or as a ready-styled CSS package	A quick start and a consistent look in designs and prototypes	No behavior or only superficial behavior, no stability commitments for code
Pattern library	A collection of patterns that combine components into recurring solutions, for example form layout, search result or empty state	Answers to the question "How can this task be solved?"	Requires components and does not replace them
Headless library	Components or state machines with behavior and accessibility, but without styling	Keyboard, focus and ARIA semantics for complex widgets such as combobox, menu or dialog	Styling, tokens, documentation and brand remain your job

Headless libraries deserve a closer look, because they are a separate option in chapter 02. *Headless* means: The library provides the behavior, and you provide the look. React Aria from Adobe describes itself as style-free and aligns semantics and keyboard behavior with the W3C ARIA Authoring Practices Guide. Zag.js provides state machines for patterns such as accordion, menu and dialog, whose APIs are fully headless and unstyled.

For web components, Lion from ING is the best-known example. Lion sees itself as a white-label layer on top of Lit that you extend into your own component layer, and it ships only functional base styles. The terms overlap: A white-label library delivers finished custom elements without brand styling, while a state machine delivers only logic that you build into elements yourself.

> **A design system describes how your products should look and behave. The component library is the part that keeps this promise in code.**

PITFALL **Mistaking the UI kit for the library**

A common mistake at the start: The design team delivers a careful Figma library, and management considers the design system done. The product teams then rebuild every component themselves, with their own keyboard handling, their own bugs and their own maintenance effort. The result is the same inconsistency as before, only with a nicer template. Make clear early on that only delivered, versioned code enforces consistency in the applications.

Why this handbook relies on web components

Web components are a bundle of browser standards. *Custom elements* define your own HTML elements such as `ds-button`. *Shadow DOM* encapsulates the internal markup and limits which nodes CSS selectors reach. Inherited properties and CSS custom properties, on the other hand, pass through the host, and documented parts open targeted styling points. *Slots* are named openings through which an element takes in content from the outside.

A library on this basis runs in every framework that can render HTML elements, and just as well without any framework. This independence is exactly why the handbook works with web components throughout. Companies switch frameworks less often than expected, but they almost always run several at the same time. A library that runs in only one of them covers part of the applications and forces the other teams to rebuild it.

Lit is a small library that simplifies writing such elements. It adds reactivity, declarative templates and utilities that reduce boilerplate, and by its own account weighs about 5 KB (minified and compressed). Every Lit component is a native custom element. Since October 2025, Lit has been part of the OpenJS Foundation and is governed by a Technical Steering Committee in which Adobe and Reddit are represented alongside Google.

Chapter 02 assesses whether web components are the right choice for your situation. In some cases, a component library built for a single framework is a better fit. Chapter 03 then explains the rationale for the key architectural decisions in detail.

Who this handbook is for

The handbook is for everyone who decides on, designs or builds a component library. It assumes experience with frontend development and explains basics only where a decision depends on them. The table maps the typical roles to their questions and the matching chapters.

Role	Typical question	What the handbook provides
CTO, head of department	Is a library of our own worth it, and what does it cost over five years?	Decision tree, cost calculation with explicit assumptions, business case and exit strategy (chapters 02, 14 and 15)
Frontend lead, architecture	How do we define component, token and package boundaries so that they remain useful for many years?	Fundamental decisions with options and consequences, code in Lit 3 and TypeScript (chapters 03 to 09)
Technical product ownership	What comes when, and how can we tell that the library is being adopted?	Roadmap with milestones and acceptance criteria, governance and metrics (chapters 14 and 15)
Library development team	How is a component built, tested, documented and published correctly?	Patterns, code examples, checks and pipeline (chapters 04 to 13)
Design and accessibility	How do decisions from Figma and from audit reports become binding in code?	Token architecture, theming and accessibility in the component (chapters 05 to 07)

What you should bring

- Confident use of TypeScript, HTML and CSS, including CSS custom properties.
- Basic knowledge of at least one widely used framework such as Angular, React or Vue.
- Experience with npm packages, a package registry and a CI pipeline.
- Knowledge of Lit is helpful but not required. Chapters 03 and 04 introduce what you need for the examples.

What this handbook is not

- **Not a Lit tutorial.** The official documentation at lit.dev explains the library completely. The handbook shows how to use Lit in a library that many teams use for years.
- **Not legal advice.** Chapters 02 and 07 name legal frameworks such as the German Accessibility Strengthening Act (BFSG). They do not replace a legal review of your individual case.
- **Not a framework comparison.** The handbook takes the frameworks of your product teams as given and shows how a library serves all of them equally well.
- **Not product advertising.** The technical chapters treat Mosaik as a case study because its architecture is publicly documented. Where another solution fits better, the handbook says so explicitly. The personal afterword on the last page reflects André Wehlert's own view.

IN PRACTICE **Start with a real question**

Read the handbook with a concrete decision in mind that is due within the next three months. Typical examples are “Build or adopt?”, “Shadow DOM for all components?” or “How do the packages reach teams outside the corporate network?”

For each chapter you read, write down which option you choose and why. At the end you have your first decision records, which chapter 14 describes as Architecture Decision Records (ADR). This saves weeks later, because new team members read the reasons instead of negotiating them again.

Three reading paths through the book

Hardly anyone reads a handbook of this size from cover to cover. That is why it offers three paths, each starting with chapter 01 and leading to a tangible result. You can combine paths. In small teams, where the same person decides and builds, that is even the norm.

Reading path	For whom	Core chapters in this order	Result after reading
Decision	CTO, head of department, budget owner, technical product owner	01, 02, 15, 14, 16, appendix A	A well-founded choice between building, adapting and buying, with a cost estimate, team setup and roadmap
Architecture	Frontend lead, architecture owner	01, 03, 04, 05, 06, 07, 08, 09, 13, appendix A	Sound fundamental decisions, recorded as ADRs, and a package structure for all frameworks
Implementation	Library development team, design, accessibility	01, 03, 04, 05, 06, 07, 10, 11, 12, 13, appendix A	Components that are tested, documented, accessible and published

On the implementation path, you first build the reference project (section “The reference project”) and set up the TypeScript and package structure with chapter 03 before you work through the component examples from chapter 04 on. If the pilot application uses a framework, also read the matching section in chapter 08 before connecting it.

The following overview shows every chapter and its importance for each path. “Core” means: Read thoroughly and compare with your own project. “Read” means: Read in context to know the reasoning. “Skim” means: Summary, boxes and checklists are enough.

Ch.	Title	Decision	Architecture	Implementation
01	How to use this handbook	Core	Core	Core
02	Build your own library, adapt, or buy?	Core	Read	Skim
03	Architecture and fundamental decisions	Skim	Core	Core
04	The component model	Skim	Core	Core
05	Design tokens and theming	Skim	Core	Core
06	Styling and customization	Skim	Core	Core
07	Accessibility	Read	Core	Core
08	One element for all frameworks	Read	Core	Read
09	The foundation: services beyond components	Skim	Core	Read
10	Quality: tests and checks	Skim	Read	Core

Ch.	Title	Decision	Architecture	Implementation
11	Documentation for humans and machines	Skim	Read	Core
12	Tools around the library	Skim	Read	Core
13	Publish, version, distribute	Skim	Core	Core
14	Operations and governance	Core	Read	Read
15	Roadmap: your own library in 12 months	Core	Read	Read
16	Case study: Mosaik	Core	Read	Read
A	Checklists	Core	Core	Core
B	Glossary	As needed	As needed	As needed

> The chapters build on each other but can be read individually. Where a chapter requires knowledge from another, it refers to it by chapter number. The glossary in appendix B explains all technical terms in a few sentences.

How each chapter is structured

Each chapter begins with a short lead that names its content and benefit. Four to eight sections follow, and they appear in the table of contents. At the end there is a summary with the key messages. The sources against which volatile information was checked are listed by chapter in appendix C.

Checklists appear at the end of a section or chapter. Appendix A bundles the most important checkpoints into thematic checklists so that you can use them directly in reviews and acceptance. If you have little time, read the lead, summary and checklist of each chapter, and you have the backbone of the argument.

Recommended starting point by situation

- **You are at the very beginning.** Read chapter 02 completely, then chapter 15. This tells you whether building a library pays off and what the first twelve months look like.
- **You have a library that is not being adopted.** Start with chapter 14 (governance, measuring adoption) and chapter 11 (documentation). Lack of acceptance rarely has technical causes.
- **You have a library that runs in only one framework.** Read chapters 03 and 08. They show how a framework-independent core with generated wrappers is built and how you migrate to it.
- **You want to remediate an existing library.** Start with the section “Remediation of an existing library” in chapter 15. It sets out a structured approach to the inventory, target API and gradual migration, with links to the chapters that explain the details.
- **You are facing an accessibility audit.** Read chapters 07 and 10. They show which requirements belong in the component and how you safeguard them automatically.

The reference project

The handbook comes with a working reference project that implements the architecture of the chapters on a small scale. It is delivered as the ZIP file `The-Blueprint-Building-a-Component-Library_Reference-Project.zip` next to the PDF and contains a monorepo with pnpm workspaces: design tokens in DTCG format with light and dark themes, the elements `ds-button`, `ds-switch` and `ds-text-field` in Lit 3, the Custom Elements Manifest, wrappers for React, Angular, Vue and Svelte generated from it, one starter for each of these frameworks, component tests in the browser, a CI pipeline and a release workflow with Changesets.

Where a code example shows a file of the project, both match. Where it deliberately shortens or simplifies, the text says so, for example for the styles in chapter 04 or the wrapper generator in chapter 08. Two differences apply throughout. For brevity, the examples register their elements with `@customElement`, while the project separates class and registration, as chapter 03 recommends for published packages. And the version number 2.4 in the examples stands for a library in operation, while the packages of the project start at 0.1.0.

Part of the project	Chapter
Monorepo with pnpm workspaces, package boundaries, `tsconfig.base.json`, `defineElement`, **entry modules under** `src/define`, `vite.config.ts`, `exports` **and** `sideEffects`	03 Architecture and fundamental decisions
API of `ds-button` **and** `ds-switch`**, form participation via ElementInternals, cancelable** `ds-change` **event**	04 The component model
Tokens in three levels, DTCG 2025.10, resolver file, `terrazzo.config.ts`**, light and dark themes, theme switch in the starters**	05 Design tokens and theming
`shared-styles.ts`, `ds-button.styles.ts`**, parts and custom properties**	06 Styling and customization
`ds-text-field` **with label, hint, error message, validation and submitting with Enter**	07 Accessibility
Wrapper generator, wrappers for React, Angular, Vue and Svelte, validator for Angular, type declarations, import without a browser	08 One element for all frameworks
Tests with Vitest Browser Mode, axe-core and the Chromium accessibility tree, `ci.yml`	10 Quality: tests and checks
Custom Elements Manifest and analyzer configuration	11 Documentation for humans and machines
Starters for React, Angular, Vue and Svelte	12 Tools around the library
Changesets, package format, `release.yml`	13 Publish, version, distribute
Repository structure, template `docs/remediation-inventory.csv`	15 Roadmap

Build and try it out

You need Node.js 22.22.3 or later in the 22.x release line, 24.15 or later in the 24.x release line, or 26 and later, as Angular 22 requires, and pnpm 11. After `pnpm install`, `pnpm --filter @acme/ds-elements exec playwright install chromium` downloads the browser for the tests once. `pnpm build` builds all packages and the four starters, then `pnpm test`, `pnpm typecheck` and `pnpm check:generated` run. The `README.md` file in the project lists all commands, the structure and the same mapping to the chapters as the table above.

- **Implementation path:** Build the project before you read chapter 03, and open the files from the table for each chapter. A starter in your own framework shows how the library feels inside an application.
- **Architecture path:** Check the package boundaries, `exports`, the generator with its exceptions file and the workflows against your own standards. Every convention in the project is an option that you confirm or change as an ADR.
- **Decision path:** The starter in the framework of the pilot application is enough to see what a library built from one core means for a product team.

> The project is deliberately small. It contains no icon package, no Tailwind bridge, no tests in Firefox and WebKit, no visual regression, no wrapper tests and no documentation website, and the README lists these gaps with the chapter that describes them. The scope `@acme`, the registry and the repository address are placeholders. You may use the code in your own software just like the code examples (appendix D).

The Mosaik case study

A handbook about architecture stays abstract as long as it does not show a real library. That is why Mosaik accompanies the whole book. Mosaik is a framework-independent component library by André Wehlert, built on Lit and publicly documented at mosaik.breadstone.de. At the time of this handbook it carries the version number 0.1.189.

347
UI elements according to the showcase, including sub-elements

4
Framework integrations: Angular, React, Vue, Svelte

10,000+
Design tokens in the reference theme Joy

140,000+
Icon definitions from 42 libraries

As of November 15, 2026, version 0.1.189

HOW MOSAIK DOES IT **One core, five targets**

Mosaik delivers native web components and generates typed wrappers for Angular, React, Vue and Svelte from them. The basis is the manifest data of the Lit components. The website documents 278 component pages in 17 categories and more than 10,000 design tokens, based on the reference theme Joy. Four themes are available: Joy, Memphis, Cosmopolitan and Soho.

For AI assistants, Mosaik publishes an llms.txt, an llms-full.txt and a Markdown file per element generated from the source code. In addition there is a public MCP server with 32 tools that can be reached without an API key.

As of November 15, 2026, version 0.1.189

Why Mosaik

Three characteristics make Mosaik a rich case study. First, its breadth: Mosaik shows not only components but also the foundation behind them, that is, router, dependency injection, portals, drag and drop, breakpoints, animations, internationalization and forms. Chapter 09 explains why a mature library needs such services.

Second, generation: The wrappers for four frameworks are created from the manifest data of the components instead of being maintained by hand. For Angular, for example, this means standalone components with signal inputs, and form controls can be connected to Angular Forms via ControlValueAccessor. Chapter 08 shows the principle. Third, machine readability: Mosaik documents itself for humans and AI tools alike, a topic of chapter 11.

André Wehlert has worked as a frontend engineer for about twenty years, from WinForms and WPF through AngularJS and Angular to web components. In each of these eras, a library of his own emerged. The Angular library was never published because it was tied to one framework. This experience led to the decision for web components and Lit.

How the handbook uses Mosaik

- The case study relies on the publicly available documentation, examples and interfaces of Mosaik, above all the website and the llms.txt (as of November 15, 2026, version 0.1.189). The main repository is private. Boxes, number blocks and tables with Mosaik figures state this date explicitly, and architectural conclusions are marked as assessments. Internal details and internal metrics do not appear.
- Mosaik appears in separate boxes titled “How Mosaik does it”. This keeps it clear what is a general recommendation and what is a concrete example.
- Chapter 16 brings the case study together as a coherent account.
- Mosaik is an example, not a benchmark. A library with 25 carefully built components can be exactly right for your company.

PITFALL **Making the scope of the case study your starting goal**

Mosaik's figures describe a state that has been reached, not a starting scope. If you set a catalog of this breadth as the goal for the first year, you spread your effort over too many fronts and end up with many half-finished components. Chapter 15 therefore recommends about 25 components up to the first production application and explains the order.

Conventions in this book

To make examples fit together across all chapters, the handbook follows a few fixed conventions. They are chosen deliberately and can be adopted in your own projects. Where you deviate, record the deviation once and then apply it just as consistently.

Prefix ds- for all general examples

All general examples use the prefix `ds-` for "design system", that is, `ds-button` with the class `DsButton`. According to the HTML specification, the name of a custom element must start with a lowercase letter, contain a hyphen and must not contain uppercase letters. A prefix satisfies this rule and prevents name conflicts with other libraries on the same page.

Choose a short, unique prefix for your library that does not sound like a department or product that might be renamed. The element name `mosaik-*` appears in the book only where it is explicitly about Mosaik, for example in the tour in chapter 16, which shows the Mosaik interfaces themselves.

Lit 3, TypeScript and decorators

The Lit examples use TypeScript and Lit 3, version 3.3.3 at the time of checking. They use decorators in the common style, that is, `@customElement` for registration and `@property` for public properties. This notation requires the TypeScript option `experimentalDecorators`. Lit additionally requires `useDefineForClassFields` to be set to `false`. The reference project registers the elements through separate entry modules instead (chapter 03).

JSON — tsconfig.json (excerpt)

```
{
  "compilerOptions": {
    "target": "ES2022",
    "module": "ESNext",
    "moduleResolution": "bundler",
    "experimentalDecorators": true,
    "useDefineForClassFields": false,
    "strict": true,
    "declaration": true
  }
}
```

Lit also supports the standard decorators. They require the keyword `accessor` on every decorated field. Chapter 03 weighs both variants. To read the examples, it is enough to know that every property marked with `@property` is part of the public API and triggers a re-render when it changes.

The following example shows all conventions at once. It is deliberately small and yet complete: registration, a public property, styling via design tokens, a documented part and a slot.

TYPESCRIPT ds-badge.ts

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

/**
 * Short status label, e.g. for "New" or "Error".
 * @slot - Label text
 * @csspart base - Outer container
 * @cssprop --ds-badge-bg - Background color
 */
@customElement('ds-badge')
export class DsBadge extends LitElement {
  static styles = css`
    :host { display: inline-flex; }
    :host([hidden]) { display: none; }
    [part='base'] {
      --_bg: var(--ds-color-feedback-neutral-bg);
      padding: 0 var(--ds-space-2);
      border-radius: var(--ds-radius-full);
      background: var(--ds-badge-bg, var(--_bg));
      font-size: var(--ds-font-label-sm-size);
      line-height: var(--ds-font-label-sm-line-height);
    }
    :host([variant='danger']) [part='base'] {
      --_bg: var(--ds-color-feedback-danger-bg);
    }
  `;

  @property({ reflect: true }) variant: 'neutral' | 'danger' = 'neutral';

  render() {
    return html`<span part="base"><slot></slot></span>`;
  }
}

declare global {
  interface HTMLElementTagNameMap {
    'ds-badge': DsBadge;
  }
}
```

Three details are typical of the whole book. The analyzer of the Custom Elements Manifest reads the JSDoc comments with `@slot`, `@csspart` and `@cssprop`, so the documentation is generated from the code (chapter 11). Colors, spacing and typography come exclusively from design tokens with the prefix `--ds-` (chapter 05). Extending `HTMLElementTagNameMap` ensures that TypeScript knows the element with the correct type for `document.querySelector('ds-badge')`.

Boxes and highlights

Element	Meaning	How to use it
In practice	Experience and proven approaches from projects	Adopt it as a template for your own approach
Pitfall	A typical mistake and its consequences	Check it against your current plan before it is implemented
How Mosaik does it	A publicly documented example from the case study	Read it as one possible implementation, not the only one
Decision	A comparison of two to four options with use case, advantages and disadvantages	Use it as the basis for a decision record (ADR)

Element	Meaning	How to use it
Checklist	Checkpoints to tick off	Use them in reviews, acceptance and planning meetings. The most important ones are collected in appendix A
Key point	The core message of a section in one sentence	Quote it in presentations and guidelines

Estimates, sources and date of checking

The handbook does not invent metrics, studies or success rates. Where a number is needed to make a decision tangible, for example for costs or team sizes, it is explicitly marked as an **estimate**. The assumptions behind it are stated openly so that you can replace them with your own values.

Volatile information such as version numbers, license status, specifications and laws was checked against the sources in appendix C as of November 15, 2026. Libraries, browsers and standards keep evolving. Check version information again before every important decision. Information that was checked later states its date in the text. The reference project is built and tested with the versions from its `pnpm-lock.yaml`, as of November 15, 2026.

Code examples and the reference project are tested in Chromium 153 on Linux and macOS, with Node.js 22 and 24. The README of the reference project contains a matrix that separates environments that were built and type-checked from those actually tested in the browser. Firefox, Safari (WebKit), Windows and use with screen readers are not tested. Test these environments yourself before you use components in production. Chapters 07 and 10 show how to add further browsers to the tests and how to plan screen reader testing.

Before you read on

- [ ] You have chosen a reading path or deliberately combined two paths.
- [ ] You have written down a concrete decision that is due within the next three months.
- [ ] You know the number of your applications and the frameworks they are built with.
- [ ] You know who in the company decides on the budget and team setup for the library.
- [ ] You have a provisional prefix for your elements that does not name a department or product.

SUMMARY

- ✓ A component library is versioned, delivered code with a documented API and built-in behavior, not a UI kit and not a collection of snippets.
- ✓ Design system, UI kit, pattern library and headless library complement the component library, but they do not replace it.
- ✓ Three reading paths (decision, architecture, implementation) each lead to a tangible result, and appendix A bundles the most important checkpoints into thematic checklists.
- ✓ Mosaik serves as the case study throughout. The starting point is the public documentation, figures carry a date, conclusions are marked as assessments, and Mosaik is an example rather than a benchmark.
- ✓ The examples of the reference project use the prefix ds-, Lit 3 and TypeScript with decorators, and the tour in chapter 16 uses the Mosaik interfaces. Estimates are marked as such.

Total cost over five years: an estimate

The total cost of ownership (TCO) does not compare license prices with salaries. It compares the entire effort for basic user interface building blocks in the company, that is, in the library and in the product teams. Only then does it become visible that a cheap option can be expensive if it shifts work to the product teams.

The following calculation is an **estimate** with explicit assumptions. It shows the mechanics, not your result. Replace every assumption with your own values, in particular the daily rate and the remaining effort.

Assumption	Value in the sample calculation	Rationale and note
Internal full cost per person-day	EUR 800	Assumption including workplace and overhead, replace the value with the figure provided by your finance team
Productive days per person-year (PY)	200	Assumption after deducting vacation, sick leave, training and organization
Cost per person-year	EUR 160,000	Results from 200 times EUR 800
Applications	8, in two frameworks	Typical situation of a mid-sized company with several product teams
License seats	30	Everyone who develops with the components
Commercial license	EUR 1,800 per seat per year	Assumption based on Vaadin's list price for the Pro tier of EUR 149 per seat per month
Effort per application without a shared library	0.5 PY per year	Assumption for custom buttons, forms, dialogs, tables and their accessibility
Remaining effort per application with an adopted or purchased library	0.2 PY per year	Assumption for gaps, brand customization and workarounds, from the second year on
Remaining effort per application with your own library	0.1 PY per year	Assumption because gaps are closed centrally, from the second year on
Period under review	5 years	Year 1 is for initial development. During this year, applications continue to work as before

For each option, the calculation assumes the library's initial development in year 1 and its annual operating effort in the following four years, in person-years. Open source requires 1.0 PY for initial development and 0.75 PY per year for the theme, facade, upgrades and support. A commercial library requires 0.5 PY for initial development and 0.5 PY per year for operations. A headless base requires 3.5 PY and 2.0 PY, respectively. Building your own requires 5.0 PY and 2.5 PY, respectively, based on the team setup in chapter 15.

Option	Library (initial development and ongoing operations)	Licenses	Product teams	Total over 5 years
Without a shared library (today's situation)	EUR 0	EUR 0	EUR 3,200,000	**EUR 3,200,000**
Adopt open source	EUR 640,000	EUR 0	EUR 1,664,000	**EUR 2,304,000**
Buy commercially	EUR 400,000	EUR 270,000	EUR 1,664,000	**EUR 2,334,000**
Own library on a headless base	EUR 1,840,000	EUR 0	EUR 1,152,000	**EUR 2,992,000**

Option	Library (initial development and ongoing operations)	Licenses	Product teams	Total over 5 years
Own library from scratch	EUR 2,400,000	EUR 0	EUR 1,152,000	**EUR 3,552,000**

The “Product teams” column counts one year with full effort and four years with remaining effort for each application. With an adopted library, that is 8 times (0.5 plus 4 times 0.2) PY, that is 10.4 PY or EUR 1,664,000. The licenses result from 30 seats times EUR 1,800 times five years.

Under these assumptions, adopting and buying are the cheapest options with eight applications. Your own library on a headless base lies just below today’s situation, while building everything yourself lies above it. This is an honest result: With eight applications and five years, your own library cannot be justified by cost alone.

Where the break-even lies

More interesting than the totals is the question of how many applications it takes for your own library to become cheaper than an adopted one. For this purpose, the table varies the period and the remaining effort with an adopted library. A remaining effort of 0.3 PY stands for a brand that differs considerably from the library’s default.

Scenario	Minimum applications for a headless base to cost less than open source	Minimum applications for building your own library to cost less than open source
5 years, remaining effort 0.2 PY	19 applications	28 applications
5 years, remaining effort 0.3 PY	10 applications	14 applications
8 years, remaining effort 0.2 PY	17 applications	24 applications
8 years, remaining effort 0.3 PY	9 applications	12 applications

The strongest lever is the remaining effort of the product teams. If it rises from 0.2 to 0.3 PY per application per year, the number of applications from which your own library pays off roughly halves. Measure this value before you decide. The two-day spike and an analysis of recent sprints provide more reliable figures than any assumption.

The following script reproduces the calculation. It is suitable for trying out your own values and making the results traceable in the business case.

TYPESCRIPT tco.ts

```
interface Option {
  name: string;
  setupPY: number;        // person-years in the first year
  runPY: number;          // person-years per subsequent year
  licensePerSeat: number; // euros per seat and year, 0 for open source
  residualPY: number;     // residual effort per app and subsequent year
}

const PY = 200 * 800;     // productive days times daily rate (assumption)
const BASE_PY = 0.5;      // effort per app without a shared library

function totalCost(o: Option, apps: number, seats: number, years: number): number {
  const library = (o.setupPY + (years - 1) * o.runPY) * PY;
  const licenses = o.licensePerSeat * seats * years;
  const productTeams = apps * (BASE_PY + (years - 1) * o.residualPY) * PY;
  return Math.round(library + licenses + productTeams);
}

const options: Option[] = [
```

```
  { name: 'Open source', setupPY: 1, runPY: 0.75, licensePerSeat: 0, residualPY: 0.2 },
  { name: 'Commercial', setupPY: 0.5, runPY: 0.5, licensePerSeat: 1800, residualPY: 0.2 },
  { name: 'Headless base', setupPY: 3.5, runPY: 2, licensePerSeat: 0, residualPY: 0.1 },
  { name: 'Custom build', setupPY: 5, runPY: 2.5, licensePerSeat: 0, residualPY: 0.1 }
];

for (const o of options) {
  console.log(o.name.padEnd(16), totalCost(o, 8, 30, 5).toLocaleString('de-DE'), 'EUR');
}
```

PITFALL **Comparing only license prices and salaries**

The most common mistake in a business case: EUR 54,000 in licenses per year is set against two salaries, and the license seems to win clearly. What remains invisible are the workarounds in eight product teams, the upgrades with breaking changes and the accessibility defects that nobody is allowed to fix. Always include the effort of the product teams, even if it is harder to measure.

> Not included are the effort for the design library in Figma, the migration of existing applications and the opportunity cost of a late start. Add these items if they carry weight in your situation. For converting existing screens, chapter 15 gives a rule, and chapter 13 covers migration guides and codemods.

Business case and risks

A business case for a component library is convincing if it does three things. It describes today’s situation with measurements instead of impressions. It discloses the assumptions so that the finance team can check them. And it divides the investment into stages after which a new decision is made.

Structure of the business case

1. **Starting situation with measurements.** Number of applications and frameworks, number of parallel implementations of the same component, findings from accessibility audits, measured remaining effort of the product teams.
2. **Goals and non-goals.** What the library should achieve and what explicitly not, for example “no business components in the first year”.
3. **Options with total cost.** At least two options with the same calculation logic and the same assumptions.
4. **Benefits beyond cost.** An accessibility fix takes effect in all applications at the same time, new applications start with finished building blocks, and a brand relaunch becomes a theme change.
5. **Risks and countermeasures.** For each option, with early warning signs.
6. **Stages and decision points.** Approval initially for phases 1 and 2 of the roadmap, a new decision after month 2 and after month 6 based on the milestones from chapter 15.

Do not promise percentages you cannot prove. Instead, define the baseline values now that you will measure again later: time to the first form screen of a new application, number of findings per accessibility audit, number of different button implementations in the code. Chapter 14 shows how to collect such values.

IN PRACTICE **Staged approval instead of a large budget**

Approving twelve months at once creates mistrust and expectations at the same time. Approval in two stages is more successful: first the foundation and core components up to the first production application, then expansion and operations. After each stage there is a tangible result, and management decides based on facts instead of slides.

Controller, mixin or base class

Reactive controller

Behavior without its own public API, especially connecting external resources such as observers, media queries, timers or keyboard navigation.

+ Several instances per element possible
+ Testable without an element, no inheritance chain
+ No influence on prototype and manifest
- Cannot declare properties or attributes on the element
- Access to the element's internals only via references passed in

Mixin

Cross-cutting public API that must look the same on many elements, such as `disabled` or `size`.

+ Properties with decorators appear in the manifest and in the types
+ Uniform behavior across all elements
- Typing requires a separate interface and a cast
- Long chains make debugging more difficult and complicate the ordering of super calls

Shared base class

A few truly universal basics such as a style reset, development warnings or support for `dir`.

+ Easy to understand
+ One place for library-wide basics
- Quickly grows into a dumping ground for everything
- Every change hits all elements at once

State and form participation

Controlled and uncontrolled

An uncontrolled element manages its state itself, like a native `input`. The application sets at most an initial value, later reads the value and listens to events. A controlled element, on the other hand, always shows exactly the value that the application specifies and reports change requests only as events, as React teams are used to.

For web components, a model that allows both has proven itself. The element updates its state itself on interaction and reports it with a cancelable event. If the application calls `preventDefault()`, the element reverts the change. If you want control, you set the value yourself after every event. If not, you ignore the event.

- The current value is a property such as `value` or `checked`, the initial value an attribute. Native elements distinguish between attribute and property in exactly the same way.
- Programmatic changes do not fire an event, but interactions do.
- The element remembers the initial value so that it can return to it when the form is reset.

PITFALL **Loops in two-way bindings**

If an element fires a `ds-change` on every property change, a two-way binding in Angular or Vue writes the value back, and that fires another event. In the best case this means duplicate work, in the worst case a loop that blocks the browser. The rule to fire events only on interaction reliably prevents this.

Form participation with ElementInternals

A native input field in the shadow root is invisible to the surrounding form. It is not submitted, not validated and not reset. Form-associated custom elements solve this. With `static formAssociated = true` and a call to `attachInternals()`, the element itself becomes a form control.

The `ElementInternals` object offers `setFormValue()` for the submitted value, `setValidity()` for validation and access to `form`, `labels`, `validity` and `states`. In addition there are four callbacks that the browser calls.

- `formAssociatedCallback(form)` when associated with a form or when that association is removed.

- `formDisabledCallback(disabled)` when the disabled state changes, including through a disabled `fieldset`.
- `formResetCallback()` after the form is reset.
- `formStateRestoreCallback(state, mode)` when the browser restores the state, with `mode` set to `"restore"`, or sets it via autofill, with `"autocomplete"`.

Form-associated custom elements have been Baseline Widely available since September 27, 2025, with Safari 16.4 in March 2023 as the last core browser to add them. For a library with Baseline Widely available as its baseline, they can therefore be used without a polyfill. ARIA reflection on `Element` and `ElementInternals`, with which an element sets its default role, has also been Widely available since April 24, 2026.

TYPESCRIPT ds-switch.ts

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

export type DsSwitchChangeEvent = CustomEvent<{ checked: boolean }>;

/**
 * Switch with form participation.
 *
 * @summary Turns a setting on or off immediately.
 * @slot - Label, also the accessible name.
 * @csspart track - The track.
 * @csspart thumb - The thumb.
 * @fires {DsSwitchChangeEvent} ds-change - After toggling, cancelable
 * @cssstate checked - The switch is on
 * @cssprop --ds-switch-track-bg - Track surface in the off state
 * @cssprop --ds-switch-track-bg-checked - Track surface in the on state
 * @cssprop --ds-switch-thumb-bg - Thumb surface
 */
@customElement('ds-switch')
export class DsSwitch extends LitElement {
  static formAssociated = true;

  static styles = css`
    :host {
      display: inline-flex; gap: var(--ds-space-2); align-items: center; cursor: pointer;
    }
    :host([hidden]) { display: none; }
    :host(:disabled) { cursor: not-allowed; opacity: 0.5; }
    [part='track'] {
      display: inline-flex; inline-size: 2.25rem; padding: 0.125rem; border-radius: 1rem;
      background: var(--ds-switch-track-bg, var(--ds-color-border-strong));
    }
    [part='thumb'] {
      inline-size: 1rem; block-size: 1rem; border-radius: 50%;
      background: var(--ds-switch-thumb-bg, var(--ds-color-bg-surface));
      transition: translate 150ms;
    }
    :host(:state(checked)) [part='track'] {
      background: var(--ds-switch-track-bg-checked, var(--ds-color-action-primary-bg));
    }
    :host(:state(checked)) [part='thumb'] { translate: 1.25rem 0; }
    :host(:dir(rtl):state(checked)) [part='thumb'] { translate: -1.25rem 0; }
    :host(:focus-visible) [part='track'] {
      outline: var(--ds-focus-ring-width, 2px) solid var(--ds-color-focus-ring);
    }
    @media (prefers-reduced-motion: reduce) { [part='thumb'] { transition: none; } }
    /* Forced colors: an edge for the track, system colors for thumb and state */
    @media (forced-colors: active) {
      [part='track'] { border: 1px solid ButtonText; }
      [part='thumb'] { background: ButtonText; }
      :host(:state(checked)) [part='track'] { background: Highlight; border-color: Highlight; }
      :host(:state(checked)) [part='thumb'] { background: HighlightText; }
```

```
      :host(:disabled) [part='track'] { background: Canvas; border-color: GrayText; }
      :host(:disabled) [part='thumb'] { background: GrayText; }
      :host(:focus-visible) [part='track'] { outline-color: Highlight; outline-offset: 2px; }
    }
  `;

  private readonly internals = this.attachInternals();
  private defaultChecked = false;

  /** On. Deliberately not reflected as an attribute, see the custom state checked. */
  @property({ type: Boolean }) checked = false;
  /** Disables the switch. A disabled fieldset has the same effect. */
  @property({ type: Boolean, reflect: true }) disabled = false;
  /** Required: the form is only valid once the switch is on. */
  @property({ type: Boolean }) required = false;
  /** Form value in the on state. */
  @property() value = 'on';
  /** Effective state including a disabled fieldset, reported by the browser. */
  @state() private formDisabled = false;

  /** Validity as with native fields, e.g. for framework wrappers. */
  get validity(): ValidityState { return this.internals.validity; }
  /** Validation message as with native fields, empty for a valid value. */
  get validationMessage(): string { return this.internals.validationMessage; }

  constructor() {
    super();
    // Without @internal, analyzer 0.11.0 lists a public field role here
    /** @internal */
    this.internals.role = 'switch';
    this.addEventListener('click', () => this.toggle());
    this.addEventListener('keydown', (event) => {
      if (event.key !== ' ' && event.key !== 'Enter') return;
      event.preventDefault();
      this.toggle();
    });
  }

  connectedCallback() {
    super.connectedCallback();
    this.defaultChecked = this.hasAttribute('checked');
    if (!this.hasAttribute('tabindex')) this.tabIndex = 0;
  }

  render() {
    return html`<span part="track"><span part="thumb"></span></span><slot></slot>`;
  }

  updated() {
    this.syncInternals();
  }

  /** Syncs form value, validity, ARIA state and custom state. */
  private syncInternals() {
    this.internals.setFormValue(this.checked ? this.value : null, String(this.checked));
    this.internals.ariaChecked = String(this.checked);
    this.internals.ariaDisabled = String(this.formDisabled);
    if (this.checked) this.internals.states.add('checked');
    else this.internals.states.delete('checked');
    const missing = this.required && !this.checked;
    this.internals.setValidity(
      missing ? { valueMissing: true } : {},
      missing ? 'Please turn on this option.' : undefined,
    );
  }
```

```
  private toggle() {
    if (this.disabled || this.formDisabled) return;
    const previous = this.checked;
    this.checked = !previous;
    this.syncInternals(); // The ds-change handler already sees the new form value
    const accepted = this.dispatchEvent(new CustomEvent('ds-change', {
      detail: { checked: this.checked }, bubbles: true, composed: true, cancelable: true,
    }));
    if (!accepted) {
      this.checked = previous;
      this.syncInternals();
    }
  }

  formDisabledCallback(disabled: boolean) {
    this.formDisabled = disabled;
  }

  formResetCallback() {
    this.checked = this.defaultChecked;
  }

  formStateRestoreCallback(state: string | File | FormData | null) {
    this.checked = state === 'true';
  }
}

declare global {
  interface HTMLElementTagNameMap {
    'ds-switch': DsSwitch;
  }
}
```

Behind the example are five decisions that apply to every form-associated element.

- `checked` is not reflected. As with native elements, the attribute remains the initial value, and styling uses the custom state `checked`, from the outside as `ds-switch:state(checked)`.
- The element sets role and ARIA states via `ElementInternals`. This creates no additional attributes on the host, and the application's attributes can still override the defaults.
- Synchronizing form value, validity, ARIA state and custom state happens in `syncInternals()`. `updated()` calls it after every update, deliberately there and not in `willUpdate()`, because `updated()` does not run during server-side rendering and these values only matter in the browser. `toggle()` additionally calls it immediately, so that a `ds-change` handler already reads the new form value. Programmatic changes to `checked`, on the other hand, only reach the form with the next update, which is why tests wait for `updateComplete`.
- The browser reports the disabled state via `formDisabledCallback()`, even when a surrounding `fieldset` is disabled. That is why `toggle()` checks this state in addition to its own property, and `:host(:disabled)` covers both cases in CSS.
- If a listener calls `preventDefault()`, `toggle()` restores the value from before the click, just as a native checkbox does, and not the opposite of the current value. The state change, event dispatch and rollback all happen synchronously before the browser paints, so nothing flickers, and Lit renders only once.

Two details concern the manifest and the wrappers. Without the comment `/** @internal */`, analyzer 0.11.0 reads the assignment to `this.internals.role` in the constructor as a public field `role` with the default value `'switch'`, and every generated wrapper would get a `role` property. The getters `validity` and `validationMessage` expose validity to the outside like a native field, and chapter 08 uses them for the validator in Angular.

In forced colors mode, the browser replaces the backgrounds of track and thumb with the page's background color, so without a countermeasure both would be invisible. The block with `forced-colors: active` therefore gives the track a border and the thumb the system color `ButtonText`. The on state additionally gets `Highlight`, the disabled state `GrayText`, and the focus ring sits with a gap next to the track. The browser keeps system colors that the element sets itself in this mode. Chapters 06 and 07 explain the rules for this.

Focus and lifecycle

Two focus models

In the first model, a native element in the shadow root is focusable, and the host forwards focus with `delegatesFocus`. This fits when the component encapsulates a native control, like `ds-button`. A click on the host or a call to `focus()` then lands on the first focusable element inside.

In the second model, the host itself is the control, as with `ds-switch`. It gets `tabindex="0"` and a role via `ElementInternals`. Set `tabindex` only in `connectedCallback()` and only if the application has not already set a value. A custom element must not add attributes in its constructor. Otherwise `document.createElement()`, for example, fails.

- Composite widgets such as lists, menus or tabs have exactly one tab stop. Inside, the arrow keys navigate, implemented with roving tabindex, that is, exactly one element with `tabindex="0"` and all others with `-1`. This logic belongs in a reactive controller.
- Focus rings use `:focus-visible`, so they appear when the browser determines that focus should be visible, typically during keyboard use. They are defined as tokens so that themes can adjust them.
- Focus is never set programmatically on load. Dialogs are an exception: They take focus when they open and return it when they close.

The lifecycle in Lit

Lit adds a reactive update cycle to the platform's callbacks. When a property changes, Lit schedules an update as a microtask, so that several changes end up in one pass. Every method of the cycle has a clear task, and most mistakes happen when work sits in the wrong method.

Method	Task	During SSR	Typical mistake
`constructor()`	Initialize fields, `attachInternals()`, listeners on the host itself	Yes	Setting attributes or accessing `window` and the document
`connectedCallback()`	External listeners, observers and subscriptions, always with a call to `super`	No	Assuming it runs only once, although elements are moved
`disconnectedCallback()`	Tear down everything from `connectedCallback()`	No	Listeners on `window` or `document` stay attached and keep the element in memory
`willUpdate(changed)`	Compute derived values, normalize inputs	Yes	Accessing the DOM although rendering is still pending
`render()`	Return the template, free of side effects	Yes	Sorting or filtering large lists on every call
`firstUpdated(changed)`	One-time work on the finished DOM, for example measuring or focus	No	Setting properties and thereby forcing a second update immediately
`updated(changed)`	Follow-up work on the DOM, `ElementInternals`, synchronization with third-party libraries	No	Changing reactive properties, which triggers another cycle