pb powerbunch



LESEPROBE

Die Blaupause

Komponentenbibliothek bauen

Architektur, Design-Tokens, Barrierefreiheit, Frameworks und Betrieb einer eigenen Komponentenbibliothek, mit Mosaik als Fallbeispiel

16 Kapitel | Codebeispiele in Lit 3 | Checklisten und Glossar | Fallstudie Mosaik

Originalseiten aus dem Handbuch mit 278 Seiten: Inhalt, Kapitel 01 sowie Auszüge aus Kapitel 02 und 04. Die übrigen Seiten sind ausgelassen, die Seitenzahlen gelten für das ganze Handbuch.

Version 1.0 · Stand November 2026 · powerbunch GmbH · Fallbeispiel: Mosaik von André Wehlert

Inhalt

So lesen Sie die Kästen

AUS DER PRAXIS
Erfahrungswissen und bewährtes Vorgehen.

STOLPERSTELLE
Typische Fehler und ihre Folgen.

SO MACHT ES MOSAIK
Belegte Lösungen aus dem Fallbeispiel.

01 Wie Sie dieses Handbuch nutzen 7

Was eine Komponentenbibliothek ist 7
Für wen dieses Handbuch ist 9
Drei Lesepfade durch das Buch 10
Das Referenzprojekt 11
Das Fallbeispiel Mosaik 13
Konventionen in diesem Buch 14

02 Eigene Bibliothek, anpassen oder kaufen? 18

Die Frage richtig stellen 18
Fünf Entscheidungskriterien 19
Die realistischen Optionen 21
Gesamtkosten über fünf Jahre: Eine Schätzrechnung 24
Business Case und Risiken 26
Exit-Strategie: Den Ausstieg vor dem Einstieg planen 27

03 Architektur und Grundentscheidungen 31

Warum Web Components die gemeinsame Basis sind 31
Lit 3 als Basis und die Alternativen 32
Shadow DOM, Light DOM oder beides 34
Repository, Pakete und Abhängigkeiten 35
Browserbasis festlegen 38
Namen, Präfixe und die Custom Element Registry 39
TypeScript und Decorators 41
Build, Module und Tree Shaking 42

04 Das Komponentenmodell 46

Die öffentliche API als Vertrag 46
Attribute und Properties 47
Events 49
Slots, Unterelemente und Parts 50
Wiederverwendbares Verhalten mit Controllern und Mixins 52
Zustand und Formularbeteiligung 54
Fokus und Lebenszyklus 58

Leistung und Definition of Done 59

05 **Design-Tokens und Theming** 62

Was Tokens sind und was nicht 62
Drei Ebenen: Basis, Semantik, Komponente 63
Das DTCG-Format 2025.10 64
Die Token-Pipeline von JSON zu CSS, TypeScript und Tailwind 67
Farbrampen, OKLCH und Kontrast 70
Themes und Farbschemata zur Laufzeit 72
Komponenten-Tokens, Dichte, Form und Typografie 74
Abgleich mit Figma 76

06 **Styling und Anpassbarkeit** 78

Kapselung durch Shadow DOM und ihre Folgen 78
Die öffentliche Styling-API als Vertrag 79
Varianten, Größen und Dichte 82
Light DOM, Tailwind und globale Stile 84
Constructable Stylesheets und Leistung 85
Fokus, Hochkontrast und Schreibrichtung 87
Versionierung der Styling-API 88
Typische Fehler und Checkliste 89

07 **Barrierefreiheit** 92

Warum die Bibliothek der wirksamste Ort ist 92
Muster aus dem ARIA Authoring Practices Guide 94
Shadow DOM: Fallen und Lösungen 95
Tastatur und Fokus 98
Fokusverwaltung in Overlays 100
Formulare: Beschriftung, Fehler, Validierung 102
Bewegung, Kontrast und erzwungene Farben 106
Testen und dokumentieren 108

08 **Ein Element für alle Frameworks** 111

Eine Quelle, viele dünne Adapter 111
Das Custom Elements Manifest als Vertrag 112
Architektur eines Wrapper-Generators 114
React: Native Unterstützung, trotzdem Wrapper 117
Angular: Standalone Components, Signal Inputs und Formulare 119
Vue 3 und Svelte 5 124
Typdefinitionen und serverseitiges Rendern 127
Wrapper testen und abnehmen 129

09 Das Fundament: Dienste jenseits der Komponenten 131

Was in die Bibliothek gehört und was in die Anwendung 131
Overlays, Portale und Positionierung 133
Internationalisierung 136
Dependency Injection und Router 139
Drag & Drop barrierefrei 142
Layout und Bewegung: Breakpoints, Container Queries, Animationen 143
Icons: Registry, Laden nach Bedarf und Lizenzen 145
Entscheidungshilfe: Gehört das in die Bibliothek? 147

10 Qualität: Tests und Prüfungen 149

Teststrategie für eine Bibliothek 149
Komponententests im echten Browser 150
Interaktion, Barrierefreiheit und Cross-Browser 153
Visuelle Regression 155
API-Stabilität und Typen 157
Wrapper, Pakete und Budgets 158
Die CI-Pipeline 160
Definition of Done und typische Fehler 161

11 Dokumentation für Menschen und Maschinen 164

Doku-Arten und ihre Leser 164
Die API-Referenz aus dem Manifest 165
Beispiele, Playground und Doku-Plattform 167
Schreibrichtlinien: Beschreibungen mit Bedeutung 169
Dokumentation für KI-Assistenten 172
Ein MCP-Server für das Design-System 174
Qualitätssicherung der KI-Doku 177

12 Werkzeuge rund um die Bibliothek 180

Welche Werkzeuge sich wann lohnen 180
Playground je Komponente 181
Themes: Editor, Generator, Paletten, Kontrast und Audit 184
Token-Bridge, Motion und Typografie 186
Visueller Designer, Rezepte und Starter-Vorlagen 187
Icon-Galerie und KI-Kontext-Generator 190
Reihenfolge und Dogfooding 191

13 Veröffentlichen, versionieren, verteilen 194

Semantic Versioning für Komponentenbibliotheken 194
Changesets: Versionen und Changelogs im Monorepo 196
Das Paketformat: ES-Module, exports, Typen, Manifest 198

Die Wahl der Registry 200
CDN-Auslieferung und Importmaps 201
Deprecation, Migrationsleitfäden und Codemods 202
Release Notes, Pipeline und Herkunftsnachweis 204

14 Betrieb und Governance 208

Teammodelle und Rollen 208
Beiträge und Entscheidungen 210
Roadmap, Priorisierung und Support 213
Adoption messen, ohne Zahlen zu erfinden 214
Kommunikation: Changelog, Release Notes und Sprechstunde 216
Finanzierung und interne Verrechnung 217
Lebenszyklus von Komponenten 218

15 Fahrplan: In 12 Monaten zur eigenen Bibliothek 221

Der Fahrplan im Überblick 221
Sanierung einer vorhandenen Bibliothek 222
Phase 1, Monat 1 bis 2: Fundament 224
Phase 2, Monat 3 bis 6: Kernkomponenten 226
Phase 3, Monat 7 bis 9: Ausbau und Frameworks 228
Phase 4, Monat 10 bis 12: Reife und Betrieb 229
Die ersten 25 Komponenten in begründeter Reihenfolge 230
Teamaufstellung und kleinere Teams 232

16 Fallstudie Mosaik 235

Steckbrief 235
Entstehung: Vier Epochen und eine Lehre 237
Die tragenden Entscheidungen 238
Was andere Teams übernehmen können und was nicht 241
Offene Punkte aus Käufersicht 244
Lehren 245

A Checklisten 247

Bereit für eine eigene Bibliothek 247
Architekturentscheidungen 247
Neue Komponente: Definition of Done 248
Design-Tokens 249
Barrierefreiheit je Komponente 250
Framework-Wrapper 250
Release 251
Deprecation 251
Dokumentation 252

Governance 252

B Glossar 254

C Quellen 260

D Lizenzbedingungen und Herausgeber 271

KAPITEL 01

Wie Sie dieses Handbuch nutzen

Dieses Kapitel klärt die Begriffe, zeigt drei Lesepfade durch das Buch und stellt das Referenzprojekt und das Fallbeispiel Mosaik vor. Danach wissen Sie, welche Kapitel Sie für Ihre Rolle zuerst lesen und wie Sie Code, Kästen und Schätzungen einordnen.

01 Was eine Komponentenbibliothek ist
02 Für wen dieses Handbuch ist
03 Drei Lesepfade durch das Buch
04 Das Referenzprojekt
05 Das Fallbeispiel Mosaik
06 Konventionen in diesem Buch

Was eine Komponentenbibliothek ist

Eine Komponentenbibliothek ist eine versionierte Sammlung wiederverwendbarer Bausteine der Benutzeroberfläche, die als Code ausgeliefert wird. Jeder Baustein hat eine dokumentierte öffentliche Schnittstelle aus Attributen, Properties, Ereignissen, Slots und Styling-Haken. Produktteams binden die Bibliothek als Paket ein und bauen damit ihre Oberflächen. Das Wort „versioniert“ ist dabei entscheidend: Eine Bibliothek ist ein Produkt mit Releases, Changelog und Zusagen zur Stabilität, kein Ordner mit kopierbaren Schnipseln.

Die Nielsen Norman Group beschreibt eine Komponentenbibliothek als Sammlung vorab festgelegter, wiederverwendbarer UI-Elemente. Jeder Eintrag umfasst typischerweise Name, Beschreibung, Attribute, Zustände und Code. Dieses Handbuch schärft die Definition für die Praxis um drei Merkmale, an denen Sie eine echte Bibliothek von einer Sammlung guter Absichten unterscheiden.

- **Ausgeliefert als Paket.** Teams installieren eine bestimmte Version und aktualisieren bewusst. Quelltext zu kopieren und lokal anzupassen ist kein Bezugsweg, sondern der Anfang einer Abspaltung.
- **Verhalten inklusive.** Fokusführung, Tastaturbedienung, Zustände wie „deaktiviert“ oder „lädt“ und die Barrierefreiheit gehören zur Komponente. Eine Anwendung soll sie nicht nachrüsten müssen.
- **Öffentliche Schnittstelle als Vertrag.** Was dokumentiert ist, gilt als zugesagt: Ab Version 1.0 bleibt es nach Semantic Versioning innerhalb einer Hauptversion kompatibel, bei Versionen unter 1.0 gelten die ausdrücklich veröffentlichten Zusagen. Was nicht dokumentiert ist, etwa interne Klassen oder die Struktur im Shadow DOM, darf sich jederzeit ändern.

Abgrenzung zu verwandten Begriffen

In Projekten fallen fünf Begriffe oft durcheinander. Die Verwechslung ist teuer, weil jeder Begriff ein anderes Budget, ein anderes Team und andere Erfolgskriterien nach sich zieht. Die folgende Tabelle legt fest, wie das Handbuch die Begriffe verwendet. Sie ist keine Normierung, sondern eine Arbeitsdefinition, die Sie in Ihrem Unternehmen übernehmen oder bewusst anders festlegen können.

Begriff	Was es ist	Was es liefert	Typische Lücke
Design-System	Gesamtheit aus Gestaltungsprinzipien, Regeln, Design-Tokens, Komponenten, Mustern, Dokumentation und dem Team, das alles pflegt	Einheitlichkeit über Produkte hinweg und eine gemeinsame Sprache zwischen Design und Entwicklung	Ohne Komponentenbibliothek bleibt es ein Regelwerk, das jedes Team selbst umsetzt
Komponentenbibliothek	Versionierte, als Paket ausgelieferte Code-Bausteine mit dokumentierter API	Lauffähige Bausteine inklusive Verhalten, Zuständen und Barrierefreiheit	Ohne Design-System fehlen die Regeln, wann welche Komponente passt

Begriff	Was es ist	Was es liefert	Typische Lücke
UI-Kit	Sammlung gestalteter Elemente, meist als Designdatei (bspw. eine Figma-Bibliothek) oder als fertig gestaltetes CSS-Paket	Schnellen Start und einheitliche Optik in Entwürfen und Prototypen	Kein oder nur oberflächliches Verhalten, keine Stabilitätszusagen für Code
Pattern Library	Sammlung von Mustern, die Komponenten zu wiederkehrenden Lösungen verbinden, bspw. Formularaufbau, Suchergebnis oder Leerzustand	Antworten auf die Frage „Wie lässt sich diese Aufgabe lösen?“	Setzt Komponenten voraus und ersetzt sie nicht
Headless-Bibliothek	Komponenten oder Zustandsmaschinen mit Verhalten und Barrierefreiheit, aber ohne Gestaltung	Tastatur, Fokus und ARIA-Semantik für komplexe Widgets wie Combobox, Menü oder Dialog	Gestaltung, Tokens, Dokumentation und Marke bleiben Ihre Aufgabe

Headless-Bibliotheken verdienen einen genaueren Blick, weil sie in Kapitel 02 eine eigene Option sind. *Headless* heißt: Die Bibliothek liefert das Verhalten, Sie liefern das Aussehen. React Aria von Adobe beschreibt sich als stilfrei und richtet Semantik und Tastaturverhalten am ARIA Authoring Practices Guide des W3C aus. Zag.js liefert Zustandsmaschinen für Muster wie Accordion, Menü und Dialog, deren APIs vollständig headless und ungestylt sind.

Für Web Components ist Lion von ING das bekannteste Beispiel. Lion versteht sich als White-Label-Schicht auf Lit, die Sie zu Ihrer eigenen Komponentenebene erweitern, und bringt nur funktionale Grundstile mit. Die Begriffe sind fließend: Eine White-Label-Bibliothek liefert fertige Custom Elements ohne Markenoptik, eine Zustandsmaschine liefert nur Logik, die Sie selbst in Elemente einbauen.

> **Ein Design-System beschreibt, wie Ihre Produkte aussehen und sich verhalten sollen. Die Komponentenbibliothek ist der Teil, der dieses Versprechen als Code einlöst.**

STOLPERSTELLE **Das UI-Kit für die Bibliothek halten**

Ein häufiger Startfehler: Das Designteam liefert eine sorgfältige Figma-Bibliothek, und die Leitung hält das Design-System damit für fertig. Die Produktteams bauen anschließend jede Komponente selbst nach, mit eigener Tastaturbedienung, eigenen Fehlern und eigenem Pflegeaufwand. Das Ergebnis ist dieselbe Uneinheitlichkeit wie vorher, nur mit schönerer Vorlage. Klären Sie früh, dass erst ausgelieferter, versionierter Code die Einheitlichkeit in den Anwendungen durchsetzt.

Warum dieses Handbuch auf Web Components setzt

Web Components sind ein Bündel von Browser-Standards. *Custom Elements* definieren eigene HTML-Elemente wie `ds-button`. *Shadow DOM* kapselt das interne Markup und begrenzt, welche Knoten CSS-Selektoren treffen. Vererbbare Eigenschaften und CSS Custom Properties wirken dagegen über den Host hinein, und dokumentierte Parts öffnen gezielte Styling-Punkte. *Slots* sind benannte Öffnungen, über die ein Element Inhalte von außen aufnimmt.

Eine Bibliothek auf dieser Basis läuft in jedem Framework, das HTML-Elemente rendern kann, und ebenso ganz ohne Framework. Genau diese Unabhängigkeit ist der Grund, warum das Handbuch durchgehend mit Web Components arbeitet. Unternehmen wechseln Frameworks seltener als gedacht, betreiben aber fast immer mehrere gleichzeitig. Eine Bibliothek, die nur in einem davon läuft, deckt einen Teil der Anwendungen ab und zwingt die übrigen Teams zum Nachbau.

Lit ist eine kleine Bibliothek, die das Schreiben solcher Elemente vereinfacht. Sie ergänzt Reaktivität, deklarative Templates und Funktionen gegen Boilerplate und wiegt nach eigener Angabe rund 5 KB (minifiziert und komprimiert). Jede Lit-Komponente ist ein natives Custom Element. Seit Oktober 2025 gehört Lit zur OpenJS

Foundation und wird von einem Technical Steering Committee gesteuert, in dem neben Google auch Adobe und Reddit vertreten sind.

Ob Web Components für Ihre Lage die richtige Wahl sind, prüft Kapitel 02 nüchtern. Es gibt Fälle, in denen die Komponentenbibliothek eines einzelnen Frameworks die bessere Antwort ist. Kapitel 03 begründet anschließend die Grundentscheidungen der Architektur im Detail.

Für wen dieses Handbuch ist

Das Handbuch richtet sich an alle, die über eine Komponentenbibliothek entscheiden, sie entwerfen oder sie bauen. Es setzt Erfahrung mit Frontend-Entwicklung voraus und erklärt Grundlagen nur dort, wo eine Entscheidung von ihnen abhängt. Die Tabelle ordnet die typischen Rollen ihren Fragen und den passenden Kapiteln zu.

Rolle	Typische Frage	Was das Handbuch liefert
CTO, Bereichsleitung	Lohnt sich eine eigene Bibliothek, und was kostet sie über fünf Jahre?	Entscheidungsbaum, Kostenrechnung mit offenen Annahmen, Business Case und Exit-Strategie (Kapitel 02, 14 und 15)
Frontend-Lead, Architektur	Wie lassen sich Komponenten, Tokens und Pakete so schneiden, dass sie viele Jahre tragen?	Grundentscheidungen mit Optionen und Folgen, Code in Lit 3 und TypeScript (Kapitel 03 bis 09)
Technische Produktverantwortung	Was kommt wann, und woran ist zu erkennen, dass die Bibliothek angenommen wird?	Fahrplan mit Meilensteinen und Abnahmekriterien, Governance und Kennzahlen (Kapitel 14 und 15)
Entwicklungsteam der Bibliothek	Wie wird eine Komponente richtig gebaut, getestet, dokumentiert und veröffentlicht?	Muster, Codebeispiele, Prüfungen und Pipeline (Kapitel 04 bis 13)
Design und Barrierefreiheit	Wie werden Entscheidungen aus Figma und aus Prüfberichten verbindlich im Code?	Token-Architektur, Theming und Barrierefreiheit in der Komponente (Kapitel 05 bis 07)

Was Sie mitbringen sollten

- Sicheren Umgang mit TypeScript, HTML und CSS, einschließlich CSS Custom Properties.
- Grundwissen über mindestens ein verbreitetes Framework wie Angular, React oder Vue.
- Erfahrung mit npm-Paketen, einer Paket-Registry und einer CI-Pipeline.
- Lit-Kenntnisse sind hilfreich, aber nicht nötig. Kapitel 03 und 04 führen ein, was Sie für die Beispiele brauchen.

Was dieses Handbuch nicht ist

- **Kein Lit-Tutorial.** Die offizielle Dokumentation unter lit.dev erklärt die Bibliothek vollständig. Das Handbuch zeigt, wie Sie Lit in einer Bibliothek einsetzen, die viele Teams über Jahre nutzen.
- **Keine Rechtsberatung.** Kapitel 02 und 07 nennen gesetzliche Rahmenbedingungen wie das Barrierefreiheitsstärkungsgesetz. Sie ersetzen keine juristische Prüfung Ihres Einzelfalls.
- **Kein Framework-Vergleich.** Das Handbuch nimmt die Frameworks Ihrer Produktteams als gegeben und zeigt, wie eine Bibliothek allen gleich gut dient.
- **Keine Produktwerbung.** Mosaik dient als Fallbeispiel, weil seine Architektur öffentlich dokumentiert ist. Wo eine andere Lösung besser passt, sagt das Handbuch es ausdrücklich.

AUS DER PRAXIS **Mit einer echten Frage einsteigen**

Lesen Sie das Handbuch mit einer konkreten Entscheidung im Kopf, die in den nächsten drei Monaten ansteht. Typische Beispiele sind „Bauen oder übernehmen?“, „Shadow DOM für alle Komponenten?“ oder „Wie erreichen die Pakete Teams außerhalb des Firmennetzes?“.

Notieren Sie zu jedem gelesenen Kapitel, welche Option Sie wählen und warum. Am Ende liegen die ersten Entscheidungsprotokolle vor, die Kapitel 14 als Architecture Decision Records (ADR) beschreibt. Das spart später Wochen, weil neue Teammitglieder die Gründe nachlesen statt sie neu zu verhandeln.

Drei Lesepfade durch das Buch

Kaum jemand liest ein Handbuch dieses Umfangs von vorn bis hinten. Deshalb bietet es drei Pfade an, die jeweils mit Kapitel 01 starten und zu einem greifbaren Ergebnis führen. Sie können Pfade kombinieren. In kleinen Teams, in denen dieselbe Person entscheidet und baut, ist das sogar der Normalfall.

Lesepfad	Für wen	Kernkapitel in dieser Reihenfolge	Ergebnis nach dem Lesen
Entscheidung	CTO, Bereichsleitung, Budgetverantwortung, technische Produktverantwortung	01, 02, 15, 14, 16, Anhang A	Begründete Wahl zwischen bauen, anpassen und kaufen, mit Kostenrahmen, Teamaufstellung und Fahrplan
Architektur	Frontend-Lead, Architekturverantwortung	01, 03, 04, 05, 06, 07, 08, 09, 13, Anhang A	Tragfähige Grundentscheidungen, festgehalten als ADR, und eine Paketstruktur für alle Frameworks
Umsetzung	Entwicklungsteam der Bibliothek, Design, Barrierefreiheit	01, 03, 04, 05, 06, 07, 10, 11, 12, 13, Anhang A	Komponenten, die getestet, dokumentiert, barrierefrei und veröffentlicht sind

Im Pfad Umsetzung bauen Sie zuerst das Referenzprojekt (Abschnitt „Das Referenzprojekt“) und richten mit Kapitel 03 die TypeScript- und Paketstruktur ein, bevor Sie die Komponentenbeispiele ab Kapitel 04 nachvollziehen. Nutzt die Pilotanwendung ein Framework, lesen Sie vor dessen Anbindung zusätzlich den passenden Abschnitt in Kapitel 08.

Die folgende Übersicht zeigt jedes Kapitel und seine Bedeutung je Pfad. „Kern“ heißt: Gründlich lesen und mit dem eigenen Projekt abgleichen. „Lesen“ heißt: Im Zusammenhang lesen, um die Begründungen zu kennen. „Überfliegen“ heißt: Zusammenfassung, Kästen und Checklisten genügen.

Kap.	Titel	Entscheidung	Architektur	Umsetzung
01	Wie Sie dieses Handbuch nutzen	Kern	Kern	Kern
02	Eigene Bibliothek, anpassen oder kaufen?	Kern	Lesen	Überfliegen
03	Architektur und Grundentscheidungen	Überfliegen	Kern	Kern
04	Das Komponentenmodell	Überfliegen	Kern	Kern
05	Design-Tokens und Theming	Überfliegen	Kern	Kern
06	Styling und Anpassbarkeit	Überfliegen	Kern	Kern
07	Barrierefreiheit	Lesen	Kern	Kern
08	Ein Element für alle Frameworks	Lesen	Kern	Lesen

Kap.	Titel	Entscheidung	Architektur	Umsetzung
09	Das Fundament: Dienste jenseits der Komponenten	Überfliegen	Kern	Lesen
10	Qualität: Tests und Prüfungen	Überfliegen	Lesen	Kern
11	Dokumentation für Menschen und Maschinen	Überfliegen	Lesen	Kern
12	Werkzeuge rund um die Bibliothek	Überfliegen	Lesen	Kern
13	Veröffentlichen, versionieren, verteilen	Überfliegen	Kern	Kern
14	Betrieb und Governance	Kern	Lesen	Lesen
15	Fahrplan: In 12 Monaten zur eigenen Bibliothek	Kern	Lesen	Lesen
16	Fallstudie Mosaik	Kern	Lesen	Lesen
A	Checklisten	Kern	Kern	Kern
B	Glossar	Bei Bedarf	Bei Bedarf	Bei Bedarf

> Die Kapitel bauen aufeinander auf, sind aber einzeln lesbar. Wo ein Kapitel Wissen aus einem anderen voraussetzt, verweist es mit Kapitelnummer darauf. Das Glossar in Anhang B erklärt alle Fachbegriffe in wenigen Sätzen.

Wie jedes Kapitel aufgebaut ist

Jedes Kapitel beginnt mit einem kurzen Vorspann, der Inhalt und Nutzen nennt. Es folgen vier bis acht Abschnitte, die im Inhaltsverzeichnis erscheinen. Am Ende steht eine Zusammenfassung mit den Kernaussagen. Die Quellen, gegen die volatile Angaben geprüft wurden, stehen nach Kapiteln geordnet in Anhang C.

Checklisten stehen jeweils am Ende eines Abschnitts oder Kapitels. Anhang A bündelt die wichtigsten Prüfpunkte daraus in thematischen Checklisten, damit Sie sie in Reviews und Abnahmen direkt verwenden können. Wer nur wenig Zeit hat, liest pro Kapitel Vorspann, Zusammenfassung und Checkliste und hat damit das Gerüst der Argumentation.

Empfohlener Einstieg nach Ausgangslage

- **Sie stehen ganz am Anfang.** Lesen Sie Kapitel 02 vollständig, danach Kapitel 15. So wissen Sie, ob sich der Aufbau lohnt und wie die ersten zwölf Monate aussehen.
- **Sie haben eine Bibliothek, die nicht angenommen wird.** Beginnen Sie mit Kapitel 14 (Governance, Adoption messen) und Kapitel 11 (Dokumentation). Fehlende Akzeptanz hat selten technische Ursachen.
- **Sie haben eine Bibliothek, die nur in einem Framework läuft.** Lesen Sie Kapitel 03 und 08. Sie zeigen, wie ein framework-unabhängiger Kern mit generierten Wrappern entsteht und wie Sie dorthin migrieren.
- **Sie wollen eine vorhandene Bibliothek sanieren.** Beginnen Sie mit dem Abschnitt „Sanierung einer vorhandenen Bibliothek“ in Kapitel 15. Er ordnet Bestandsaufnahme, Ziel-API und schrittweise Migration und verweist auf die Kapitel mit den Einzelheiten.
- **Sie stehen vor einer Prüfung der Barrierefreiheit.** Lesen Sie Kapitel 07 und 10. Sie zeigen, welche Anforderungen in die Komponente gehören und wie Sie sie automatisiert absichern.

Das Referenzprojekt

Zum Handbuch gehört ein lauffähiges Referenzprojekt, das die Architektur der Kapitel in kleinem Umfang umsetzt. Es liegt als ZIP-Datei `Die-Blaupause-Komponentenbibliothek-bauen_Referenzprojekt.zip` neben dem PDF und enthält ein Monorepo mit pnpm-Workspaces: Design-Tokens im DTCG-Format mit Themes für hell

und dunkel, die Elemente `ds-button`, `ds-switch` und `ds-text-field` in Lit 3, das Custom Elements Manifest, daraus generierte Wrapper für React, Angular, Vue und Svelte, je einen Starter für diese Frameworks, Komponententests im Browser, eine CI-Pipeline und einen Release-Workflow mit Changesets.

Wo ein Codebeispiel eine Datei des Projekts zeigt, stimmen beide überein. Wo es bewusst kürzt oder vereinfacht, sagt der Text das, bspw. bei den Stilen in Kapitel 04 oder beim Wrapper-Generator in Kapitel 08. Zwei Unterschiede gelten durchgehend. Die Beispiele registrieren ihre Elemente der Kürze halber mit `@customElement`, das Projekt trennt Klasse und Registrierung, wie Kapitel 03 es für veröffentlichte Pakete empfiehlt. Und die Versionsnummer 2.4 in den Beispielen steht für eine Bibliothek im Betrieb, die Pakete des Projekts beginnen mit 0.1.0.

Teil des Projekts	Kapitel
Monorepo mit pnpm-Workspaces, Paketschnitt, `tsconfig.base.json`, `defineElement`**, Einstiegsmodule unter** `src/define`, `vite.config.ts`, `exports` **und** `sideEffects`	03 Architektur und Grundentscheidungen
API von `ds-button` **und** `ds-switch`**, Formularbeteiligung über ElementInternals, Event** `ds-change` **mit Rücknahme**	04 Das Komponentenmodell
Tokens in drei Ebenen, DTCG 2025.10, Resolver-Datei, `terrazzo.config.ts`**, Themes hell und dunkel, Theme-Schalter in den Startern**	05 Design-Tokens und Theming
`shared-styles.ts`, `ds-button.styles.ts`**, Parts und Custom Properties**	06 Styling und Anpassbarkeit
`ds-text-field` **mit Beschriftung, Hinweis, Fehlermeldung, Validierung und Absenden mit Enter**	07 Barrierefreiheit
Wrapper-Generator, Wrapper für React, Angular, Vue und Svelte, Validator für Angular, Typdeklarationen, Import ohne Browser	08 Ein Element für alle Frameworks
Tests mit Vitest Browser Mode, axe-core und dem Accessibility Tree von Chromium, `ci.yml`	10 Qualität: Tests und Prüfungen
Custom Elements Manifest und Konfiguration des Analyzers	11 Dokumentation für Menschen und Maschinen
Starter für React, Angular, Vue und Svelte	12 Werkzeuge rund um die Bibliothek
Changesets, Paketformat, `release.yml`	13 Veröffentlichen, versionieren, verteilen
Struktur des Repositorys, Vorlage `docs/remediation-inventory.csv`	15 Fahrplan

Bauen und ausprobieren

Vorausgesetzt sind Node.js 22.22.3 oder neuer in Linie 22, 24.15 oder neuer in Linie 24 oder ab 26, wie Angular 22 es verlangt, und pnpm 11. Nach `pnpm install` lädt `pnpm --filter @acme/ds-elements exec playwright install chromium` einmalig den Browser für die Tests. `pnpm build` baut alle Pakete und die vier Starter, danach laufen `pnpm test`, `pnpm typecheck` und `pnpm check:generated`. Die Datei `README.md` im Projekt nennt alle Befehle, den Aufbau und dieselbe Zuordnung zu den Kapiteln wie die Tabelle oben.

- **Pfad Umsetzung:** Bauen Sie das Projekt, bevor Sie Kapitel 03 lesen, und öffnen Sie zu jedem Kapitel die Dateien aus der Tabelle. Ein Starter im eigenen Framework zeigt, wie sich die Bibliothek in einer Anwendung anfühlt.
- **Pfad Architektur:** Prüfen Sie Paketschnitt, `exports`, den Generator mit seiner Ausnahmedatei und die Workflows gegen Ihre eigenen Vorgaben. Jede Vorgabe im Projekt ist eine Option, die Sie als ADR bestätigen oder ändern.

- **Pfad Entscheidung:** Der Starter im Framework der Pilotanwendung genügt, um zu sehen, was eine Bibliothek aus einem Kern für ein Produktteam bedeutet.

> Das Projekt ist bewusst klein. Es enthält kein Icon-Paket, keine Tailwind-Brücke, keine Tests in Firefox und WebKit, keine visuelle Regression, keine Wrapper-Tests und keine Dokumentations-Website, das README listet diese Lücken mit dem Kapitel, das sie beschreibt. Scope `@acme`, Registry und Repository-Adresse sind Platzhalter. Den Code dürfen Sie wie die Codebeispiele in eigene Software übernehmen (Anhang D).

Das Fallbeispiel Mosaik

Ein Handbuch über Architektur bleibt abstrakt, solange es keine echte Bibliothek zeigt. Deshalb begleitet Mosaik das ganze Buch. Mosaik ist eine frameworkunabhängige Komponentenbibliothek von André Wehlert auf Basis von Lit, öffentlich dokumentiert unter mosaik.breadstone.de. Zum Stand dieses Handbuchs trägt sie die Versionsnummer 0.1.189.

347	4	10.000+	140.000+
UI-Elemente einschließlich Unterelementen	Framework-Integrationen: Angular, React, Vue, Svelte	Design-Tokens im Referenztheme Joy	Icon-Definitionen aus 42 Bibliotheken

Stand 05.11.2026, Version 0.1.189

SO MACHT ES MOSAIK **Ein Kern, fünf Ziele**

Mosaik liefert native Web Components und generiert daraus typisierte Wrapper für Angular, React, Vue und Svelte. Grundlage sind die Manifestdaten der Lit-Komponenten. Die Website dokumentiert 278 Komponentenseiten in 17 Kategorien und mehr als 10.000 Design-Tokens, bezogen auf das Referenztheme Joy. Vier Themes stehen bereit: Joy, Memphis, Cosmopolitan und Soho.

Für KI-Assistenten veröffentlicht Mosaik eine llms.txt, eine llms-full.txt und eine aus dem Quellcode generierte Markdown-Datei je Element. Dazu kommt ein öffentlicher MCP-Server mit 32 Werkzeugen, der ohne API-Schlüssel erreichbar ist.

Stand 05.11.2026, Version 0.1.189

Warum gerade Mosaik

Drei Eigenschaften machen Mosaik zu einem ergiebigen Fallbeispiel. Erstens die Breite: Mosaik zeigt nicht nur Komponenten, sondern auch das Fundament dahinter, also Router, Dependency Injection, Portale, Drag & Drop, Breakpoints, Animationen, Internationalisierung und Formulare. Kapitel 09 erklärt, warum eine reife Bibliothek solche Dienste braucht.

Zweitens die Generierung: Die Wrapper für vier Frameworks entstehen aus den Manifestdaten der Komponenten, statt von Hand gepflegt zu werden. Für Angular bedeutet das bspw. Standalone Components mit Signal Inputs, Formularsteuerelemente lassen sich über ControlValueAccessor an Angular Forms anbinden. Kapitel 08 zeigt das Prinzip. Drittens die Maschinenlesbarkeit: Mosaik dokumentiert sich für Menschen und für KI-Werkzeuge gleichermaßen, ein Thema von Kapitel 11.

André Wehlert arbeitet seit rund zwanzig Jahren als Frontend-Engineer, von WinForms über WPF und AngularJS bzw. Angular bis zu Web Components. In jeder dieser Epochen entstand eine eigene Bibliothek. Die Angular-Bibliothek wurde nie veröffentlicht, weil sie an ein Framework gebunden war. Aus dieser Erfahrung folgte die Entscheidung für Web Components und Lit.

Wie das Handbuch Mosaik verwendet

- Das Fallbeispiel stützt sich auf die öffentlich zugängliche Dokumentation, die Beispiele und die Schnittstellen von Mosaik, vor allem auf die Website und die llms.txt (Stand 05.11.2026, Version 0.1.189). Das Hauptrepository ist privat. Kästen, Zahlenblöcke und Tabellen mit Mosaik-Zahlen nennen diesen Stand ausdrücklich, architektonische Schlussfolgerungen sind als Einordnung gekennzeichnet. Interna und interne Kennzahlen kommen nicht vor.
- Mosaik erscheint in eigenen Kästen mit dem Titel „So macht es Mosaik“. So bleibt klar, was allgemeine Empfehlung ist und was ein konkretes Beispiel.
- Kapitel 16 fasst das Fallbeispiel als zusammenhängende Fallstudie zusammen.
- Mosaik ist ein Beispiel, kein Maßstab. Eine Bibliothek mit 25 sorgfältig gebauten Komponenten kann für Ihr Unternehmen genau die richtige sein.

STOLPERSTELLE **Den Umfang des Fallbeispiels zum Startziel machen**

Die Zahlen von Mosaik beschreiben einen erreichten Stand, keinen Startumfang. Wer einen Katalog dieser Breite als Ziel für das erste Jahr festlegt, verteilt die Kraft auf zu viele Baustellen und liefert am Ende viele halbfertige Komponenten. Kapitel 15 empfiehlt deshalb rund 25 Komponenten bis zur ersten produktiven Anwendung und begründet die Reihenfolge.

Konventionen in diesem Buch

Damit Beispiele über alle Kapitel zusammenpassen, folgt das Handbuch wenigen festen Konventionen. Sie sind bewusst gewählt und lassen sich in eigene Projekte übernehmen. Wo Sie abweichen, sollten Sie die Abweichung einmal festhalten und dann ebenso konsequent einhalten.

Präfix ds- für alle allgemeinen Beispiele

Alle allgemeinen Beispiele verwenden das Präfix `ds-` für „Design-System“, also `ds-button` mit der Klasse `DsButton`. Der Name eines Custom Elements muss nach der HTML-Spezifikation mit einem Kleinbuchstaben beginnen, einen Bindestrich enthalten und darf keine Großbuchstaben enthalten. Ein Präfix erfüllt diese Regel und verhindert Namenskonflikte mit anderen Bibliotheken auf derselben Seite.

Wählen Sie für Ihre Bibliothek ein kurzes, eindeutiges Präfix, das nicht nach einer Abteilung oder einem Produkt klingt, die sich umbenennen könnten. Der Elementname `mosaik-*` erscheint im Buch nur, wenn es ausdrücklich um Mosaik geht.

Lit 3, TypeScript und Decorators

Die Codebeispiele sind in TypeScript mit Lit 3 geschrieben, zum Prüfstand in Version 3.3.3. Sie verwenden Decorators im verbreiteten Stil, also `@customElement` für die Registrierung und `@property` für öffentliche Eigenschaften. Diese Schreibweise setzt die TypeScript-Option `experimentalDecorators` voraus. Lit verlangt dazu, dass `useDefineForClassFields` auf `false` steht. Das Referenzprojekt registriert die Elemente stattdessen über eigene Einstiegsmodule (Kapitel 03).

JSON tsconfig.json (Auszug)

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

Lit unterstützt auch die standardisierten Decorators. Diese verlangen das Schlüsselwort `accessor` an jedem dekorierten Feld. Kapitel 03 wägt beide Varianten ab. Für das Lesen der Beispiele genügt es zu wissen, dass jede mit `@property` markierte Eigenschaft Teil der öffentlichen API ist und bei Änderung ein erneutes Rendern auslöst.

Das folgende Beispiel zeigt alle Konventionen auf einmal. Es ist bewusst klein und doch vollständig: Registrierung, öffentliche Property, Styling über Design-Tokens, ein dokumentierter Part und ein 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;
  }
}
```

Drei Details sind typisch für das ganze Buch. Die JSDoc-Kommentare mit `@slot`, `@csspart` und `@cssprop` liest der Analyzer des Custom Elements Manifest aus, die Dokumentation entsteht also aus dem Code (Kapitel 11). Farben, Abstände und Schrift kommen ausschließlich aus Design-Tokens mit dem Präfix `--ds-` (Kapitel 05). Die Erweiterung von `HTMLElementTagNameMap` sorgt dafür, dass TypeScript das Element bei `document.querySelector('ds-badge')` mit dem richtigen Typ kennt.

Kästen und Hervorhebungen

Element	Bedeutung	Wie Sie es nutzen
Aus der Praxis	Erfahrungswissen und erprobtes Vorgehen aus Projekten	Als Vorlage für Ihr eigenes Vorgehen übernehmen
Stolperstelle	Typischer Fehler mit seinen Folgen	Gegen Ihren aktuellen Plan prüfen, bevor er umgesetzt wird
So macht es Mosaik	Öffentlich dokumentiertes Beispiel aus dem Fallbeispiel	Als eine mögliche Umsetzung lesen, nicht als einzige
Entscheidung	Gegenüberstellung von zwei bis vier Optionen mit Einsatzfall, Vorteilen und Nachteilen	Als Grundlage für ein Entscheidungsprotokoll (ADR) verwenden
Checkliste	Prüfpunkte zum Abhaken	In Reviews, Abnahmen und Planungsrunden einsetzen, die wichtigsten gebündelt in Anhang A
Merksatz	Kernaussage eines Abschnitts in einem Satz	In Präsentationen und Leitlinien zitieren

Schätzungen, Quellen und Prüfstand

Das Handbuch erfindet keine Kennzahlen, Studien oder Erfolgsquoten. Wo eine Zahl nötig ist, um eine Entscheidung greifbar zu machen, etwa bei Kosten oder Teamgrößen, ist sie ausdrücklich als **Schätzung** gekennzeichnet. Die Annahmen dahinter stehen offen daneben, damit Sie sie durch Ihre eigenen Werte ersetzen können.

Volatile Angaben wie Versionsnummern, Lizenzstände, Spezifikationen und Gesetze wurden zum Stand 05.11.2026 gegen die Quellen in Anhang C geprüft. Bibliotheken, Browser und Normen entwickeln sich weiter. Prüfen Sie Versionsangaben deshalb vor jeder wichtigen Entscheidung erneut. Angaben, die erst später geprüft wurden, nennen ihr Datum im Text. Das Referenzprojekt ist mit den Versionen aus seiner `pnpm-lock.yaml` gebaut und getestet, Stand 03.11.2026.

Codebeispiele und Referenzprojekt sind in Chromium 153 unter Linux und macOS geprüft, mit Node.js 22 und 24. Die README des Referenzprojekts zeigt dazu eine Matrix, die gebaute und typgeprüfte von tatsächlich im Browser geprüften Umgebungen trennt. Nicht geprüft sind Firefox, Safari (WebKit), Windows und die Bedienung mit Screenreadern. Prüfen Sie diese Umgebungen selbst, bevor Sie Komponenten produktiv einsetzen. Wie Sie weitere Browser in die Tests aufnehmen und Screenreader-Prüfungen planen, zeigen Kapitel 07 und 10.

Bevor Sie weiterlesen

- [ ] Sie haben einen Lesepfad gewählt oder bewusst zwei Pfade kombiniert.
- [ ] Sie haben eine konkrete Entscheidung notiert, die in den nächsten drei Monaten ansteht.
- [ ] Sie kennen die Zahl Ihrer Anwendungen und die Frameworks, in denen sie gebaut sind.
- [ ] Sie wissen, wer im Unternehmen über Budget und Teamaufstellung für die Bibliothek entscheidet.
- [ ] Sie haben ein vorläufiges Präfix für Ihre Elemente, das keine Abteilung und kein Produkt nennt.

DAS WICHTIGSTE IN KÜRZE

- Eine Komponentenbibliothek ist versionierter, ausgelieferter Code mit dokumentierter API und eingebautem Verhalten, kein UI-Kit und keine Schnipselsammlung.
- Design-System, UI-Kit, Pattern Library und Headless-Bibliothek ergänzen die Komponentenbibliothek, ersetzen sie aber nicht.
- Drei Lesepfade (Entscheidung, Architektur, Umsetzung) führen jeweils zu einem greifbaren Ergebnis, Anhang A bündelt die wichtigsten Prüfpunkte in thematischen Checklisten.
- Mosaik dient als durchgehendes Fallbeispiel. Ausgangspunkt ist die öffentliche Dokumentation, Zahlen tragen einen Stand, Schlussfolgerungen sind als Einordnung gekennzeichnet, und Mosaik ist Beispiel statt Maßstab.
- Alle Beispiele nutzen das Präfix ds-, Lit 3 und TypeScript mit Decorators, Schätzungen sind als solche gekennzeichnet.

SO MACHT ES MOSAIK **Mosaik als Basis prüfen**

Mosaik ist eine Bibliothek von Grund auf, mit eigenem Fundament aus Diensten, das Kapitel 09 vorstellt. Wer Mosaik nicht als Vorbild, sondern als Grundlage für die eigenen Produkte prüft, findet auf der Website drei Angaben, die in jede Bewertung gehören.

Die Version lautet 0.1.189. Nach Semantic Versioning 0.x wird die öffentliche API damit noch nicht als stabil zugesagt, über die Qualität sagt die Versionsnummer nichts. Laut Fußbereich der Website steht Mosaik unter MIT-Lizenz (Stand 05.11.2026). Die Pakete liegen auf GitHub Packages, lokal erfordert die Installation einen klassischen GitHub-PAT mit dem Recht `read:packages` sowie Zugriff auf das Paket.

Stand 05.11.2026, Version 0.1.189

Gesamtkosten über fünf Jahre: Eine Schätzrechnung

Die Gesamtkosten (Total Cost of Ownership, TCO) vergleichen nicht Lizenzpreise mit Gehältern. Sie vergleichen den gesamten Aufwand für Oberflächen-Grundbausteine im Unternehmen, also in der Bibliothek und in den Produktteams. Erst so wird sichtbar, dass eine billige Option teuer sein kann, wenn sie Arbeit in die Produktteams verschiebt.

Die folgende Rechnung ist eine **Schätzung** mit offengelegten Annahmen. Sie zeigt die Mechanik, nicht Ihr Ergebnis. Ersetzen Sie jede Annahme durch Ihre eigenen Werte, insbesondere Tagessatz und Restaufwand.

Annahme	Wert in der Beispielrechnung	Begründung und Hinweis
Interne Vollkosten je Personentag	800 Euro	Annahme inklusive Arbeitsplatz und Gemeinkosten, ersetzen Sie den Wert durch die Zahl Ihres Controllings
Produktive Tage je Personenjahr (PJ)	200	Annahme nach Abzug von Urlaub, Krankheit, Weiterbildung und Organisation
Kosten je Personenjahr	160.000 Euro	Ergibt sich aus 200 mal 800 Euro
Anwendungen	8, in zwei Frameworks	Typische Lage eines mittelgroßen Unternehmens mit mehreren Produktteams
Lizenzplätze	30	Alle Personen, die mit den Komponenten entwickeln
Lizenz kommerziell	1.800 Euro je Platz und Jahr	Annahme, angelehnt an den Listenpreis der Pro-Stufe von Vaadin von 149 Euro je Platz und Monat
Aufwand je Anwendung ohne gemeinsame Bibliothek	0,5 PJ je Jahr	Annahme für eigene Buttons, Formulare, Dialoge, Tabellen und deren Barrierefreiheit
Restaufwand je Anwendung mit übernommener oder gekaufter Bibliothek	0,2 PJ je Jahr	Annahme für Lücken, Markenanpassung und Workarounds, ab dem zweiten Jahr
Restaufwand je Anwendung mit eigener Bibliothek	0,1 PJ je Jahr	Annahme, weil Lücken zentral geschlossen werden, ab dem zweiten Jahr
Betrachtungszeitraum	5 Jahre	Jahr 1 ist Aufbau, die Anwendungen arbeiten in diesem Jahr noch wie bisher

Für die Bibliothek selbst setzt die Rechnung je Option einen Aufbau im ersten Jahr und einen Betrieb in den vier Folgejahren an, jeweils in Personenjahren. Für Open Source sind das 1,0 PJ Aufbau und 0,75 PJ Betrieb für Theme, Fassade, Upgrades und Support, für die kommerzielle Bibliothek je 0,5 PJ. Die Headless-Basis braucht 3,5 PJ und 2,0 PJ, der Eigenbau 5,0 PJ und 2,5 PJ, abgeleitet aus der Teamaufstellung in Kapitel 15.

Option	Bibliothek (Aufbau und Betrieb)	Lizenzen	Produktteams	Summe 5 Jahre
Ohne gemeinsame Bibliothek (heutiger Zustand)	0 Euro	0 Euro	3.200.000 Euro	**3.200.000 Euro**
Open Source übernehmen	640.000 Euro	0 Euro	1.664.000 Euro	**2.304.000 Euro**
Kommerziell kaufen	400.000 Euro	270.000 Euro	1.664.000 Euro	**2.334.000 Euro**
Eigene auf Headless-Basis	1.840.000 Euro	0 Euro	1.152.000 Euro	**2.992.000 Euro**
Eigene von Grund auf	2.400.000 Euro	0 Euro	1.152.000 Euro	**3.552.000 Euro**

Die Spalte „Produktteams“ rechnet für jede Anwendung ein Jahr mit vollem Aufwand und vier Jahre mit Restaufwand. Bei übernommener Bibliothek sind das 8 mal (0,5 plus 4 mal 0,2) PJ, also 10,4 PJ oder 1.664.000 Euro. Die Lizenzen ergeben sich aus 30 Plätzen mal 1.800 Euro mal fünf Jahren.

Unter diesen Annahmen sind Übernehmen und Kaufen bei acht Anwendungen die günstigsten Optionen. Eine eigene Bibliothek auf Headless-Basis liegt knapp unter dem heutigen Zustand, der vollständige Eigenbau darüber. Das ist ein ehrliches Ergebnis: Bei acht Anwendungen und fünf Jahren lässt sich eine eigene Bibliothek nicht allein über Kosten begründen.

Wo der Gleichstand liegt

Interessanter als die Summen ist die Frage, ab wie vielen Anwendungen eine eigene Bibliothek günstiger wird als eine übernommene. Die Tabelle variiert dafür den Zeitraum und den Restaufwand bei übernommener Bibliothek. Ein Restaufwand von 0,3 PJ steht für eine Marke, die deutlich vom Standard der Bibliothek abweicht.

Szenario	Headless-Basis günstiger als Open Source ab	Eigenbau günstiger als Open Source ab
5 Jahre, Restaufwand 0,2 PJ	19 Anwendungen	28 Anwendungen
5 Jahre, Restaufwand 0,3 PJ	10 Anwendungen	14 Anwendungen
8 Jahre, Restaufwand 0,2 PJ	17 Anwendungen	24 Anwendungen
8 Jahre, Restaufwand 0,3 PJ	9 Anwendungen	12 Anwendungen

Der stärkste Hebel ist der Restaufwand der Produktteams. Steigt er von 0,2 auf 0,3 PJ je Anwendung und Jahr, halbiert sich ungefähr die Zahl der Anwendungen, ab der sich eine eigene Bibliothek rechnet. Messen Sie diesen Wert deshalb, bevor Sie entscheiden. Der Zwei-Tage-Spike und eine Auswertung der letzten Sprints liefern belastbarere Zahlen als jede Annahme.

Das folgende Skript bildet die Rechnung nach. Es eignet sich, um eigene Werte durchzuspielen und die Ergebnisse im Business Case nachvollziehbar zu machen.

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');
}
```

STOLPERSTELLE **Nur Lizenzpreise und Gehälter vergleichen**

Der häufigste Fehler im Business Case: 54.000 Euro Lizenz im Jahr werden gegen zwei Gehälter gestellt, und die Lizenz gewinnt scheinbar klar. Unsichtbar bleiben die Workarounds in acht Produktteams, die Upgrades mit Breaking Changes und die Barrierefreiheitsmängel, die niemand beheben darf. Rechnen Sie immer den Aufwand der Produktteams mit, auch wenn er schwerer zu erheben ist.

> Nicht enthalten sind der Aufwand für die Designbibliothek in Figma, die Migration bestehender Anwendungen und die Opportunitätskosten eines späten Starts. Ergänzen Sie diese Posten, wenn sie in Ihrer Lage ins Gewicht fallen. Für die Umstellung bestehender Bildschirme nennt Kapitel 15 eine Regel, Migrationsleitfäden und Codemods behandelt Kapitel 13.

Business Case und Risiken

Ein Business Case für eine Komponentenbibliothek überzeugt, wenn er drei Dinge leistet. Er beschreibt den heutigen Zustand mit Messwerten statt mit Eindrücken. Er legt die Annahmen offen, sodass das Controlling sie prüfen kann. Und er teilt die Investition in Stufen, nach denen erneut entschieden wird.

Aufbau des Business Case

1. **Ausgangslage mit Messwerten.** Zahl der Anwendungen und Frameworks, Zahl der parallelen Implementierungen derselben Komponente, Befunde aus Prüfungen zur Barrierefreiheit, gemessener Restaufwand der Produktteams.
2. **Ziele und Nicht-Ziele.** Was die Bibliothek leisten soll und was ausdrücklich nicht, bspw. „keine Fachkomponenten im ersten Jahr“.
3. **Optionen mit Gesamtkosten.** Mindestens zwei Optionen mit derselben Rechenlogik und denselben Annahmen.
4. **Nutzen jenseits der Kosten.** Eine Korrektur der Barrierefreiheit wirkt in allen Anwendungen gleichzeitig, neue Anwendungen starten mit fertigen Bausteinen, ein Marken-Relaunch wird zu einer Theme-Änderung.
5. **Risiken und Gegenmaßnahmen.** Für jede Option, mit Frühwarnzeichen.
6. **Stufen und Entscheidungspunkte.** Freigabe zunächst für Phase 1 und 2 des Fahrplans, erneute Entscheidung nach Monat 2 und nach Monat 6 anhand der Meilensteine aus Kapitel 15.

Versprechen Sie keine Prozentwerte, die Sie nicht belegen können. Legen Sie stattdessen jetzt die Basiswerte fest, die Sie später erneut messen: Dauer bis zum ersten Formularbildschirm einer neuen Anwendung, Zahl der Befunde je Prüfung zur Barrierefreiheit, Zahl der unterschiedlichen Button-Implementierungen im Code. Kapitel 14 zeigt, wie Sie solche Werte erheben.

Controller, Mixin oder Basisklasse

Reactive Controller

Verhalten ohne eigene öffentliche API, besonders die Anbindung externer Ressourcen wie Observer, Medienabfragen, Timer oder Tastaturnavigation.

+ Mehrere Instanzen je Element möglich
+ Ohne Element testbar, keine Vererbungskette
+ Kein Einfluss auf Prototyp und Manifest
- Kann keine Properties oder Attribute am Element deklarieren
- Zugriff auf Interna des Elements nur über übergebene Referenzen

Mixin

Querschnittliche öffentliche API, die bei vielen Elementen gleich aussehen muss, etwa `disabled` oder `size`.

+ Properties mit Decorators erscheinen im Manifest und in den Typen
+ Einheitliches Verhalten über alle Elemente
- Die Typisierung braucht ein separates Interface und einen Cast
- Lange Ketten erschweren Fehlersuche und die Reihenfolge der super-Aufrufe

Gemeinsame Basisklasse

Wenige, wirklich universelle Grundlagen wie Stil-Reset, Entwicklungswarnungen oder die Unterstützung von `dir`.

+ Leicht zu verstehen
+ Eine Stelle für bibliotheksweite Grundlagen
- Wächst schnell zur Ablage für alles
- Jede Änderung trifft alle Elemente gleichzeitig

Zustand und Formularbeteiligung

Kontrolliert und unkontrolliert

Ein unkontrolliertes Element verwaltet seinen Zustand selbst, wie ein natives `input`. Die Anwendung setzt höchstens einen Startwert, liest später den Wert und hört auf Events. Ein kontrolliertes Element zeigt dagegen immer genau den Wert, den die Anwendung vorgibt, und meldet Änderungswünsche nur als Event, wie es React-Teams gewohnt sind.

Für Web Components bewährt sich ein Modell, das beides erlaubt. Das Element aktualisiert seinen Zustand bei einer Interaktion selbst und meldet ihn mit einem verhinderbaren Event. Ruft die Anwendung `preventDefault()` auf, nimmt das Element die Änderung zurück. Wer kontrollieren will, setzt nach jedem Event den Wert selbst, wer nicht, ignoriert das Event.

- Der aktuelle Wert ist eine Property wie `value` oder `checked`, der Startwert ein Attribut. Native Elemente unterscheiden genauso zwischen Attribut und Property.
- Programmatische Änderungen lösen kein Event aus, Interaktionen schon.
- Das Element merkt sich den Startwert, um beim Zurücksetzen des Formulars dorthin zurückzukehren.

STOLPERSTELLE **Schleifen in Zwei-Wege-Bindungen**

Feuert ein Element bei jeder Property-Änderung ein `ds-change`, schreibt eine Zwei-Wege-Bindung in Angular oder Vue den Wert zurück, und das löst erneut ein Event aus. Im besten Fall entsteht doppelte Arbeit, im schlechtesten eine Schleife, die den Browser blockiert. Die Regel, Events nur bei Interaktion auszulösen, verhindert das zuverlässig.

Formularbeteiligung mit ElementInternals

Ein natives Eingabefeld im Shadow Root ist für das umgebende Formular unsichtbar. Es wird nicht übertragen, nicht validiert und nicht zurückgesetzt. Form-associated Custom Elements lösen das. Mit `static formAssociated = true` und einem Aufruf von `attachInternals()` wird das Element selbst zum Formularsteuerelement.

Das Objekt `ElementInternals` bietet `setFormValue()` für den übertragenen Wert, `setValidity()` für die Validierung sowie Zugriff auf `form`, `labels`, `validity` und `states`. Dazu kommen vier Rückrufe, die der Browser aufruft.

- `formAssociatedCallback(form)` bei Zuordnung zu einem Formular oder deren Aufhebung.
- `formDisabledCallback(disabled)`, wenn sich der deaktivierte Zustand ändert, auch durch ein deaktiviertes `fieldset`.
- `formResetCallback()` nach dem Zurücksetzen des Formulars.
- `formStateRestoreCallback(state, mode)`, wenn der Browser den Zustand wiederherstellt, dann mit `mode` gleich `"restore"`, oder per Autofill setzt, dann mit `"autocomplete"`.

Form-associated Custom Elements sind seit dem 27.09.2025 Baseline Widely available, als letzter Kernbrowser kam Safari 16.4 im März 2023 hinzu. Für eine Bibliothek mit Baseline Widely available als Grundlinie sind sie damit ohne Polyfill nutzbar. Die ARIA-Reflexion an `Element` und `ElementInternals`, mit der ein Element seine Standardrolle festlegt, ist seit dem 24.04.2026 ebenfalls Widely available.

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;
  }
}
```

Hinter dem Beispiel stehen fünf Entscheidungen, die für jedes formularbeteiligte Element gelten.

- `checked` wird nicht reflektiert. Das Attribut bleibt wie bei nativen Elementen der Startwert, gestaltet wird über den Custom State `checked`, von außen als `ds-switch:state(checked)`.
- Rolle und ARIA-Zustände setzt das Element über `ElementInternals`. So entstehen keine zusätzlichen Attribute am Host, und Attribute der Anwendung können die Standardwerte trotzdem überschreiben.
- Der Abgleich von Formularwert, Validität, ARIA-Zustand und Custom State steht in `syncInternals()`. Nach jedem Update ruft `updated()` sie auf, bewusst dort und nicht in `willUpdate()`, weil `updated()` beim serverseitigen Rendern nicht läuft und diese Werte nur im Browser Bedeutung haben. `toggle()` ruft sie zusätzlich sofort auf, damit ein Handler von `ds-change` bereits den neuen Formularwert liest. Programmatische Änderungen von `checked` kommen dagegen erst mit dem nächsten Update im Formular an, Tests warten deshalb auf `updateComplete`.
- Den deaktivierten Zustand meldet der Browser über `formDisabledCallback()`, auch wenn ein umgebendes `fieldset` deaktiviert ist. Deshalb prüft `toggle()` diesen Zustand zusätzlich zur eigenen Property, und `:host(:disabled)` deckt im CSS beide Fälle ab.
- Ruft ein Listener `preventDefault()` auf, stellt `toggle()` den Wert von vor dem Klick wieder her, wie es auch eine native Checkbox tut, und nicht das Gegenteil des aktuellen Werts. Umschalten, Ereignis und Rücknahme laufen synchron, bevor der Browser zeichnet, deshalb flackert nichts, und Lit rendert nur einmal.

Zwei Details betreffen Manifest und Wrapper. Ohne den Kommentar `/** @internal */` liest der Analyzer 0.11.0 die Zuweisung an `this.internals.role` im Konstruktor als öffentliches Feld `role` mit dem Standardwert `'switch'`, und jeder generierte Wrapper erhielte eine Property `role`. Die Getter `validity` und `validationMessage` geben die Gültigkeit nach außen wie ein natives Feld, Kapitel 08 nutzt sie für den Validator in Angular.

Im Modus erzwungener Farben ersetzt der Browser die Hintergründe von Schiene und Schieber durch die Hintergrundfarbe der Seite, ohne Gegenmaßnahme wären beide unsichtbar. Der Block mit `forced-colors: active` gibt der Schiene deshalb einen Rahmen und dem Schieber die Systemfarbe `ButtonText`. Der eingeschaltete Zustand erhält zusätzlich `Highlight`, der deaktivierte `GrayText`, und der Fokusring steht mit Abstand neben der Schiene. Systemfarben, die das Element selbst setzt, behält der Browser in diesem Modus bei. Die Regeln dafür erklären Kapitel 06 und 07.

Fokus und Lebenszyklus

Zwei Fokusmodelle

Beim ersten Modell ist ein natives Element im Shadow Root fokussierbar, und der Host leitet den Fokus mit `delegatesFocus` weiter. Das passt, wenn die Komponente ein natives Steuerelement kapselt, wie `ds-button`. Ein Klick auf den Host oder ein Aufruf von `focus()` landet dann beim ersten fokussierbaren Element im Inneren.

Beim zweiten Modell ist der Host selbst das Steuerelement, wie bei `ds-switch`. Er erhält `tabindex="0"` und eine Rolle über `ElementInternals`. Setzen Sie `tabindex` erst im `connectedCallback()` und nur, wenn die Anwendung keinen eigenen Wert gesetzt hat. Im Konstruktor darf ein Custom Element keine Attribute hinzufügen, sonst schlägt bspw. `document.createElement()` fehl.

- Zusammengesetzte Widgets wie Listen, Menüs oder Tabs haben genau einen Tab-Stopp. Innerhalb navigieren die Pfeiltasten, umgesetzt mit Roving Tabindex, also genau einem Element mit `tabindex="0"` und allen anderen mit `-1`. Diese Logik gehört in einen Reactive Controller.
- Fokusringe erscheinen über `:focus-visible` nur bei Tastaturbedienung und sind als Token definiert, damit Themes sie anpassen können.
- Der Fokus wird nie beim Laden programmatisch gesetzt. Eine Ausnahme sind Dialoge, die beim Öffnen den Fokus übernehmen und ihn beim Schließen zurückgeben.

Der Lebenszyklus in Lit

Lit ergänzt die Rückrufe der Plattform um einen reaktiven Update-Zyklus. Ändert sich eine Property, plant Lit ein Update als Microtask, sodass mehrere Änderungen in einem Durchlauf landen. Jede Methode des Zyklus hat eine klare Aufgabe, und die meisten Fehler entstehen, wenn Arbeit in der falschen Methode steckt.

Methode	Aufgabe	Beim SSR	Typischer Fehler
`constructor()`	Felder initialisieren, `attachInternals()`, Listener am Host selbst	Ja	Attribute setzen oder auf `window` und das Dokument zugreifen
`connectedCallback()`	Externe Listener, Observer und Abonnements, immer mit Aufruf von `super`	Nein	Annahme, er laufe nur einmal, obwohl Elemente verschoben werden
`disconnectedCallback()`	Alles aus `connectedCallback()` zurückbauen	Nein	Listener an `window` oder `document` bleiben hängen und halten das Element im Speicher