DIY-FRAMEWORK FÜR SPAS

Hold my beer!

Das Wie und Warum selbst gebauter Frontend-Frameworks.

Die Liste der Dinge, die man als normalsterbliche:r Softwareentwickler:in nach Ansicht graubärtiger Senior-Nerds unter gar keinen Umständen jemals selbst programmieren sollte, ist lang. Datums- und Zeitverarbeitung? Sehr gefährlich. Kryptografie? Um Himmels willen! Ein eigenes Frontend-Framework für das neue SPA-Projekt? Klares Zeichen von Naivität und/oder Selbstüberschätzung und außerdem ein Affront gegen die natürliche Ordnung der Dinge.

Doch tief im Herzen wissen wir natürlich, dass solches Gatekeeper-Gemurmel, wie alles in der Softwareentwicklung, eher den Charakter von Richtlinien als von Regeln hat. So haben natürlich alle Entwickler:innen der aktuellen Industriestandards im Hinblick auf Zeit, Kryptografie und SPA-Frameworks irgendwann einmal ihre jeweils ersten Schritte in den jeweiligen Disziplinen unternommen – Lernen und Testen muss immer gestattet sein. Und selbst außerhalb von Experimenten muss auch ein:e Diplom-Ingenieur:in von und zu Senior-Standardlösung einsehen, dass besondere Umstände häufig an den jeweiligen Sonderfall angepasste Software erfordern. Die Frage ist dann nur: Was zählt als „besonderer Umstand"? Wann wäre es im Fall von Webfrontends an der Zeit, den etablierten Frontend-Frameworks den Rücken zuzuwenden und sich ein eigenes SPA-Framework zu schreiben?

Frontend-Frameworks im Jahr 2025

Das Webfrontend-Ökosystem lässt sich mit etwas Wohlwollen als „unübersichtlich" charakterisieren. Neben zahlreichen Klassikern wie React, Vue und Angular haben sich mittlerweile die sogenannten Full-Stack-Frameworks nach Machart von Next.js etabliert, die die APIs und Paradigmen von etwa React (im Fall von Next.js) oder Vue (im Fall von Nuxt) mit ins Backend holen (Bild 1) [1]. „Etabliert" heißt hierbei keinesfalls „fertig" – bei allen relevanten Kandidaten läuft die Dependency-Umwälzpumpe am Drehzahlbegrenzer, und ein Prozess des permanenten Neuerfindens ist Standard. Was aus der Außenperspektive wie das schiere Chaos wirkt, ergibt in der historischen Betrachtung ein gewisses Maß an Sinn.

Bild: Shutterstock / passion artist

Die Geschichte von Frontend-Frameworks beginnt mit der Feststellung, dass statische HTML-Seiten ungeeignet waren, um die hyperinteraktiven Web-Apps der Generation Web 2.0 (zum Beispiel Facebook und Twitter) abzubilden. Interaktive User Interfaces im Browser selbst waren dank JavaScript zwar prinzipiell machbar, aber aufgrund von Einschränkungen und Kompatibilitätsproblemen seitens der damaligen Browser-APIs tatsächlich niemandem zuzumuten. Unter diesen Prämissen (native Browserfeatures sind nicht zu gebrauchen und UIs sind Aufgabe des Clients) entstanden die klassischen Frontend-Frameworks und auf ihrer Basis klassische Single-Page-Apps: komplexe clientseitige JavaScript-Programme, die jeden Aspekt des Browsers abstrahieren und den Server zum reinen API-Provider degradierten. Die offensichtlichen Probleme mit diesem Ansatz sind mannigfaltig: Komplexen Content auf anämischen Antik-Androids zu rendern ist nicht vergnügungssteuerpflichtig, und die seinerzeit relevanten Suchmaschinen waren von leeren HTML-Gerüsten, die erst in einem hypothetischen Browser mit Inhalt versehen werden sollten, auch nicht begeistert. Doch zum Glück ist JavaScript, wenn man die entsprechende Geisteshaltung pflegt, wie Gewalt: Wo ein bisschen JS ein bisschen hilft, hilft ein Mehr an JS mit Sicherheit noch viel weiter.

Full-Stack-Frameworks bringen die Programmierparadigmen und Problemlösungen von Single-Page-App-Frameworks auf den Server. Mit ausreichend hohem Abstraktionsniveau ist es möglich, SPA-Frameworks sowohl in den JavaScript-Engines von Browsern als auch JS-Runtimes auf Servern lauffähig zu machen. Auf diese Weise lässt sich der Traum von „isomorphic JavaScript" verwirklichen: eine einzige UI-Beschreibung, die auf dem Server in einen statischen HTML-String übersetzt wird, auf dem Client aber ein modernes, hyperinteraktives UI rendert. Dies beschleunigt, so die Idee, das initiale Laden einer SPA durch Pre-Rendering und macht nebenher auch noch die Suchmaschinen-Crawler froh. Der Hydration genannte In-Browser-Mutationsprozess, der aus statischem HTML eine interaktive Web-App macht, ist nicht komplett trivial, da HTML eine sehr verlustbehaftete Serialisierung der JavaScript-Repräsentation einer Web-App darstellt – doch wo ein Wille ist, ist bekanntlich ein Weg.

Mehrere Variationen dieses Ansatzes werden durch steten Input aus den Geldbörsen von Big-Tech-Sugardaddys und/oder Venture Capital genährt. Diversität ist daher wirklich gegeben: Obwohl ziemlich allen Frontend-Frameworks im Jahr 2025 die bereits beschriebene Ideologie zugrunde liegt, ist die Developer Experience sehr unterschiedlich. React biegt jedes einzelne Web-API auf funktionale Programmierung um, lässt Entwickler:innen aber ansonsten viel Auswahl bei der Komposition des sonstigen Tech-Stacks. Bei Angular hat hingegen jedes Modul seinen ihm eigens zugewiesenen Platz, und einen offiziellen Compiler gibt es obendrein. Das Svelte-Framework implementiert sich konsequenterweise direkt komplett als Compiler, während Vue.js aussieht, als würde es auf normale Webstandards aufbauen – wenn auch mit Ergänzungen, die Marty McFly aus dem Jahr 2062 mitgebracht zu haben scheint.

Der Markt für JavaScript-Frameworks ist also groß (denn alle machen irgendwas mit Webtech), die Marktteilnehmer (alle, die irgendwas mit Webtech machen) sind divers, und das Gottvertrauen der Geldgeber ist unerschütterlich. Wie kann da das Produkt „JavaScript-Framework" etwas anderes sein als eine mit Höchstgeschwindigkeit rotierende Neuerfindungs-Maschine? Und wie kann ein JavaScript-Framework unter diesen Umständen etwas anderes sein als ein Produkt? Denn eines ist unübersehbar: Obwohl alle relevanten Frontend-Frameworks dank freizügigster Open-Source-Lizenzen zum Nulltarif zu haben und ohne Einschränkungen einsetzbar sind, operieren sie aus der Kundenperspektive der Entwickler:innen wie Hochglanzprodukte. Sie leisten sich durchgestylte Webseiten, richten ausschweifende Konferenzen aus und sorgen mit kurzen Produktlebenszyklen dafür, dass die Frameworks Feature-technisch auf Augenhöhe bleiben. Das alles ist per se unproblematisch: Wenn Webseiten von Open-Source-Projekten kein akutes Augenbluten auslösen, Konferenzen ein gewisses Maß an spätrömischer Dekadenz mitbringen und die Software stets bugfrei und konkurrenzfähig gehalten wird, ist das zunächst positiv. Von Big Tech oder Big VC finanziert zu werden ist ebenfalls allgemein akzeptiertes Geschäftsgebaren.

> „Das Webfrontend-Ökosystem lässt sich als ‚unübersichtlich' charakterisieren."

Doch wenn Webframeworks als Produkte in einem Markt operieren und wir Entwickler:innen die Konsumenten und Zielgruppen bilden, dürfen wir die feilgebotenen Waren objektiv und gnadenlos bewerten. Wir haben es schließlich nicht mehr mit den Hobbyprojekten kauziger Free-Software-Philosophen zu tun, sondern mit den Angeboten großer, gut kapitalisierter Organisationen. Wir dürfen und sollten uns fragen, ob das, was uns der Markt vorsetzt, für unsere jeweiligen Einsatzzwecke auch tatsächlich geeignet ist.

Welchen Job machen Frameworks überhaupt?

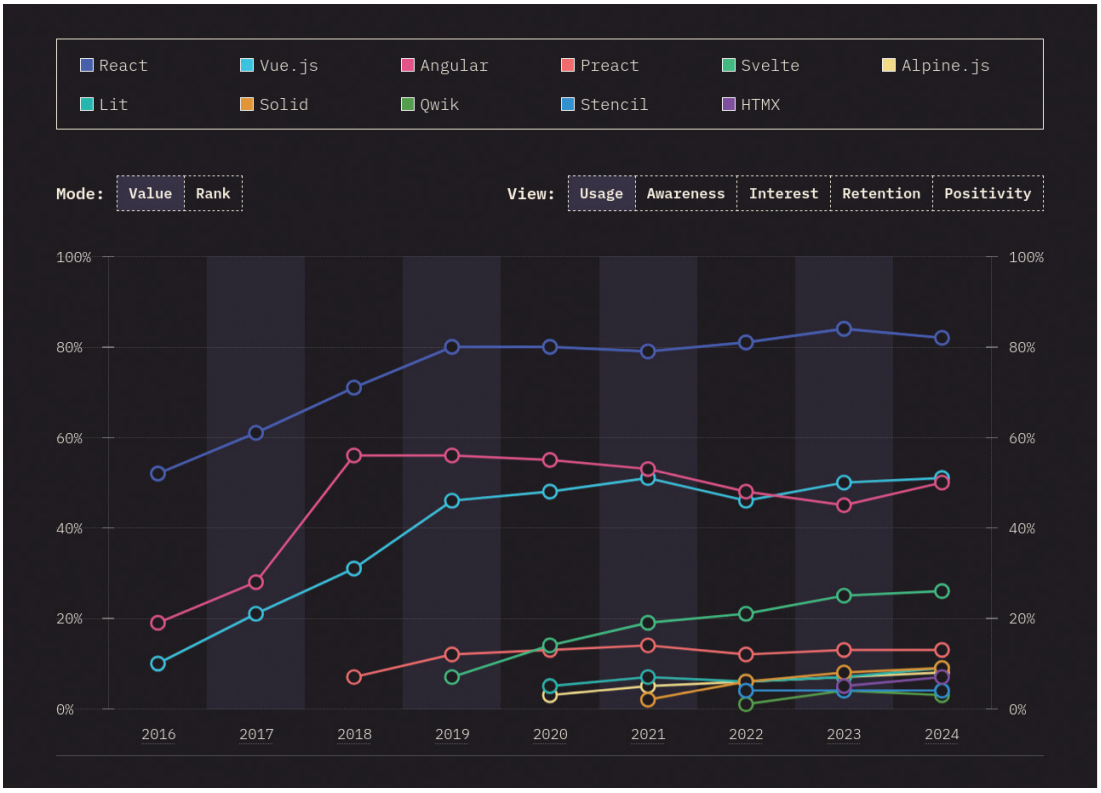
Wikipedia beschreibt Softwareframeworks nicht unzutreffend als ein „Programmiergerüst", das „den Rahmen zur Verfügung [stellt], innerhalb dessen der Programmierer eine Anwendung erstellt". Das Hauptmerkmal eines solchen Gerüsts (oder idealerweise eines jeden Gerüsts, sei es in Software, auf Baustellen oder auf Spielplätzen) sollte Stabilität sein – ein Merkmal, für das das Frontend-JavaScript-Ökosystem nicht gerade bekannt ist. Mit modernen SPA-Frameworks umgesetzte Projekte zeichnen sich oft durch extravagant dimensionierte Dependency-Trees aus, die täglich gepflegt werden wollen. Breaking Changes sind nicht selten und betreffen mal die betroffenen Frameworks selbst, mal Third-Party-Komponenten, mal Sub-Sub-Sub-Dependencies. Im günstigsten Fall verbraucht das dauerhafte Updaten nur Zeit, Budget und Lebensfreude aufseiten der Ausführenden, im schlechtesten Fall verliert das Projekt den Kampf gegen den Wildwuchs und ▶

ist zu radikalen Klagemaßnahmen gezwungen. Drakonische NDAs hindern den Autor dieser Zeilen daran, die blutigen Details über die zahlreichen (ungeplanten!) Inhouse-Framework-Forks auszuplaudern, mit denen Entwickler:innen in namhaften Firmen kämpfen müssen, doch er kann zumindest verraten, dass es keine Seltenheit ist, einen eigenen Angular-Entwicklungszweig auf Basis einer Version aus dem Frühmittelalter zu pflegen. Denn was sonst will man machen, wenn man zum Beispiel für über zehn Jahre an unveränderliche Hardware gebunden ist, neue Framework-Versionen aber annehmen, auf immer neueren und schnelleren Geräten zu laufen? Was sonst will man tun, wenn Abhängigkeiten von Abhängigkeiten kombiniert mit den schieren Ausmaßen des Projekts quartalsweise aufschlagende Breaking Changes nicht bewältigbar machen?

„Quartalsweise aufschlagende Breaking Changes" sind dabei keine Übertreibung, denn die Instabilität von Frontend-JavaScript hat viele Ausprägungen und Gründe. Manchmal ist das Gewackel Programm, wie die tatsächlich alle drei Monate stattfindenden Major-Releases von Angular. Andere Kandidaten, wie React, unterstützen alte APIs auch noch nach dem Release der Nachfolger ihrer Nachfolger lange weiter, doch das Ökosystem tendiert dazu, auf die neuen Züge umgehend aufzuspringen. Third-Party-Komponenten, Dokumentation, Tutorials und Personal (Entwickler:innen wie Influencer:innen) sind stets an der Bleeding Edge orientiert, was zu faktisch nicht begründbarem (aber auch nicht ignorierbarem) Refactoring-Druck auf ansonsten stabile Codebases führt. Ein Stabilitätsanker sieht anders aus.

Das permanente Neuerfinden ist auf den Produktcharakter der Frontend-Frameworks zurückzuführen. Neue Versionen und Innovationen füttern den News-Cycle, und auch die marginalsten Performance-Verbesserungen in Edge Cases reichen, um die Konkurrenz beim Benchmark zu überholen. Ein Vergleich mit der Autoindustrie drängt sich geradezu auf. Inkrementelle Verbesserungen an der Substanz der Produkte werden mit großem Marketing-Aufwand verkauft, und die Implementierung der inkrementellen Verbesserungen treibt die Komplexität der Produkte – was bei Kleinwagen die kilometerlangen Kabelbäume sind, sind bei modernen Frontend-Frameworks die Codezeilen, Compiler und Komplexitäten. Hinzu kommt der Wartungsaufwand eines Frontend-Framework-Produkts für seine Endnutzer (also uns Entwickler:innen) mit seinen ständigen Dependency-Updates – von dem wir noch nicht mal in allen Fällen etwas haben.

Denn wo wir bei einem Auto bei fast allen Fahrten die meisten der vier verfügbaren Räder verwenden, haben Frontend-Frameworks Ballast an Bord, der für viele Projekte irrelevant ist, aber die Komplexität treibt. Ein offensichtliches Beispiel ist, dass General-Purpose-Frameworks definitionsgemäß mehr Features an Bord haben, als ein einzelnes Projekt je verwenden könnte. Das ist nicht nur im Hinblick auf Download-Performance lästig, sondern auch der Stabilität abträglich. Breaking Changes in Updates bedeuten immer Test- und Deployment-Aufwand, selbst wenn unser spezifisches Projekt die betroffenen APIs nicht oder kaum nutzt. Außerdem sind, ihrem Übergewicht zum Trotz, alle gängigen SPA-Setups auf die in ihrem Rahmen maximal mögliche Performance gebürstet. Daran ist in Isolation auch nichts auszusetzen, doch wer schon mal versucht hat, aus einem Programm die perfekte Performance herauszuquetschen, kennt den Komplexitäts-Kipppunkt: Irgendwann ist mit Vereinfachungen des Codes



Bildquelle: 2024.stateofjs.com [1]

Die (nicht auch nur ansatzweise repräsentative) Umfrage „State of JavaScript 2024" zeigt, dass es unter den Frontend-Frameworks keinen unangefochtenen Marktführer gibt (Bild 1)

nichts mehr zu erreichen, und elaborierte Caching-Strategien und Mikrooptimierungen beginnen das einst elegante Programm zu verkomplizieren.

Ebenfalls als Ballast bewertbar sind lebenserhaltende Maßnahmen für antike APIs, die eigentlich niemand mehr verwenden möchte. Reacts Klassenkomponenten haben beispielsweise ihren Hauptwohnsitz in der Mottenkiste, sind aber weiterhin Bestandteil des Kernprodukts. Das ist nicht nur buchstäblicher Ballast in Form von nutzlosen Kilobytes, sondern verkompliziert, wie jedes Subsystem in einem Programm, durch Wechselwirkungen jedes andere Subsystem – von der Angriffsfläche für Sicherheitslücken und Dependency-Verfall ganz zu schweigen. Kein Code ist so effizient, schnell, sicher und wartungsfreundlich wie kein Code – doch „kein Code" entspricht nicht dem Ethos des modernen SPA-Stacks.

> „Stabilität ist ein Merkmal, für das das Frontend-JavaScript-Ökosystem nicht gerade bekannt ist."

Die Aufgabe, die Frontend-Frameworks im Jahr 2025 wahrnehmen, ist eine vielfältige. Sie kombinieren ausgesuchte Toolchains mit Hochleistungs-Abstraktionen über Browser-APIs und verbinden beides mit einer dazu passenden Entwicklungs- und Designphilosophie. Marketing und Community-Pflege sorgen dabei für ein passendes Ambiente. Sie sind die Bentleys der Webentwicklung: komfortabel, leistungsstark und technisch ausgefeilt. Aber sind sie (mit ihren immensen Unterhaltskosten) das passende Vehikel, um als normaler Mensch im Jahr 2025 ein ganz normales Discount-Dashboard zu bauen? Sind sie nötig? Sinnvoll? Kosteneffektiv?

In welchem Jahrhundert leben wir eigentlich wirklich?

Was Frontend-Frameworks faktisch leisten, ist das eine. Das andere ist ihr Raison d'Être, der auf zwei vermeintliche Problemstellungen zurückzuführen ist:

1. Web-Apps erfordern ein Maß an clientseitiger Interaktivität, das nur durch die Priorisierung von Frontend-JavaScript über alle anderen Überlegungen umsetzbar ist.
2. Browser haben hierfür nicht die passenden Bordmittel, weshalb massiver Abstraktions- und Engineering-Aufwand zu betreiben ist.

In den frühen 2010er-Jahren, als die aktuelle Generation der SPA-Frameworks (ihrerseits die Grundlage der zurzeit so populären Full-Stack-Frameworks) aus der Ursuppe gekrochen kam, war das eine zutreffende Zustandsbeschreibung. Dinosaurier wie der Internet Explorer waren noch weit vom Aussterben entfernt, und über 60 Prozent aller Websites diesseits der Andromeda-Galaxie sahen sich zur Verwendung von jQuery gezwungen, um zahlloser Browser-Inkonsistenzen Herr zu werden und fehlende UI-Features via Plug-in einzurüsten. Nur sind die frühen 2010er mittlerweile vorbei!

Webbrowser wurden in allen Dimensionen massiv aufgerüstet. DOM-APIs haben ein Komfortniveau erreicht, das ein Programmieren ohne Abstraktionen für Normalsterbliche möglich macht. Die Einführung von Web Components bringt ein natives Komponentenmodell in den Browser, ECMA-Script-Module machen viele Aspekte klassischer Toolchains überflüssig, HTML hat mit Formular-Features und Neuerungen wie <*dialog*> massiv aufgerüstet, und moderne CSS-Features wirken aus der Perspektive von 2013 wie Science-Fiction. Die Performance hat sich ebenfalls stark verbessert, und durch schnelleres Ausrollen von neuen Releases erreichen neue Browser-Versionen in kürzester Zeit die Massen der Nutzer:innen. Der gesamte Punkt 2 der obigen Kausalkette ist nicht mehr als eine von Opas Geschichten von der fernen Front des Browserkriegs.

Und Punkt 1 sieht bei nüchterner Betrachtung nicht viel besser aus. Es ist keinesfalls zwingend nötig, Web-Apps primär mit JavaScript zu entwickeln. Das ist nur der aktuelle Zeitgeist. Viele Use Cases (Dateneingabe, Content-Websites) sind hervorragend mit serverseitigen Mitteln abzubilden, sofern alles schnell genug lädt – was ohne den Overhead eines fetten Frontend-Frameworks problemlos machbar ist. Wenn mal etwas mehr clientseitige Interaktivität erforderlich ist, gibt es zahllose Optionen, die nicht zu 100 Prozent aus JavaScript-Frameworks bestehen. HTMX [2] etwa motzt HTML so auf, dass mit ein paar Attributen API-Requests durchgeführt und Views mit Updates versorgt werden können:

```
<!-- Wie zu Opas Zeiten einfach die Library laden -->
<script src="https://unpkg.com/htmx.org@2.0.4"></script>

<!-- Button-Klick triggert einen Post-Request. Die
  Antwort ersetzt, wenn eingetroffen, den Button
  (das "outerHTML) -->
<button hx-post="/clicked" hx-swap="outerHTML">
  Klick!
</button>
```

Hotwired [3] funktioniert hingegen wie ein SPA-Layer über eine klassische serverseitige Web-App, indem es HTML-Updates ohne Seiten-Reload in die gerade aktive Seite hineinpatcht. View Transitions [4] sind ein in fast allen Browsern verfügbarer Webstandard, der einen ähnlichen Effekt erreicht und ganz ohne Third-Party-Software auskommt. Und dank der bereits benannten Browser-Updates können moderat kompetente Entwickler:innen auch einfach ein wenig Vanilla-JS schreiben und ein paar Web Components zusammenstecken.

Lösen die genannten Alternativen zu SPA-Frameworks jedes Problem in der Frontendentwicklung? Natürlich nicht, keine Software löst alle Probleme in ihrer Domäne unter allen Umständen. Aber sie sind Alternativen. Sie lassen Endprodukte entstehen, die leichtgewichtiger, weniger komplex und weniger fragil sind als das, was der State of the Art verordnet. ▶

Moderne Webentwicklung des Jahres 2025 besteht leider überwiegend darin, unter der Annahme der Alternativlosigkeit eine hochgezüchtete JavaScript-Maximallösung durch das Realitäts-Nadelöhr zu zwängen. Das ist kein Engineering, sondern Ideologie. Engineering findet Kompromisse und lokale Maxima für die jeweiligen Anwendungsfälle. Spezialisierte Bereiche wie Raumfahrt oder Gelddruckmaschinen wie der Facebook-Feed profitieren davon, dass keine Kosten und Mühen gescheut werden, sollten aber Ausnahmen bleiben. Kosten und Mühen zu scheuen und zu reduzieren ist gutes Engineering. Vereinfachung ist gut. Weniger Wartungsaufwand ist gut. Billiger ist besser. Solange der Gegenstand unserer Engineering-Bemühungen ein brauchbares Endprodukt wird, sollten wir versuchen, das Meiste mit dem Wenigsten zu erreichen.

Hold my beer

Woraus besteht ein Softwareframework? Eigentlich nur aus Software und Entscheidungen. Es legt Trade-offs fest, definiert APIs und bestimmt, was in welcher Reihenfolge zu passieren hat. Und da wir als Softwareentwickler:innen den ganzen Tag nichts anderes machen, als APIs zu definieren und Trade-offs festzulegen, sind wir allesamt prädestiniert für die Fertigung eines eigenen Frontend-Frameworks. Wird es das brillanteste Frontend-Framework aller Zeiten werden? Vielleicht, vielleicht nicht. Es wird aber unser eigenes sein, mit den Zutaten und Entscheidungen, die uns zusagen. Fangen wir doch gleich mit ein paar Entscheidungen an!

Erste Design-Entscheidung: JavaScript depriorisieren. Das klingt nach einer für ein Frontend-Framework widersinnigen Entscheidung, aber es ist unser Framework – wir können machen, was wir wollen! Wenn wir unser Tool so viele Aufgaben wie möglich an eingebaute APIs und Standards wie HTML und CSS delegieren lassen, müssen wir weniger reimplementieren und benötigen weniger Tooling. Wir nutzen die Browser des Jahres 2025 und werden uns ihres vollen Funktionsumfangs bedienen! Das Komponentenmodell? Web Components. Styling-System? Nacktes, standardkonformes CSS. State Management? Wenn wir ohnehin Web Components mit JavaScript-Klassen machen, benötigen wir keine zusätzlichen State-Container oder Signals – und falls sich das eines Tages ändert, können wir das revidieren, denn unsere zweite Designentscheidung ist die für den radikalen Pragmatismus.

> „Das Web ist eine stabile Plattform für Content und User Interfaces aller Arten."

Es gibt keinen Grund, päpstlicher zu sein als der Papst. Wenn der Browser ein bestimmtes Feature nicht bietet, installieren wir uns eine passende Library – im Idealfall eine mit wenig Overhead und vielen Alternativen, damit bei unangenehmen Breaking Changes oder einem Ausfall der Maintainer Alternativen parat stehen. uhtml [5] ist eine Rendering-Engine auf Basis von Template Literals. Sie ist kompakt, kann viel und es existieren buchstäblich Hunderte fast identische Alternativen. Ornament [6] macht das Handling von Web Components etwas angenehmer und besteht aus lediglich einer Handvoll Funktionen, die explizit für den „Bau dein eigenes Framework"-Anwendungsfall ausgelegt sind.

Auf Basis dieser zwei Entscheidungen ist es uns bereits möglich, eine einfache Komponenten-Basisklasse zusammenzustecken, wie in Listing 1 gezeigt.

Listing 1: Eine einfache Komponenten-Basisklasse

```
// src/framework.js

import { reactive, connected, debounce } from
  "@sirpepe/ornament";
import { html, render } from "uhtml";

export class BaseElement extends HTMLElement {
  // Enthält das UI des Elements. Details unwichtig :)
  #root = this.attachShadow({ mode: "closed" });

  // Macht uhtml in Subklassen als this.html verfügbar
  html(...args) {
    return html(...args);
  }

  // Subklassen können eine Methode namens render()
  // definieren, die das UI des Elements definiert.
  // Sie wird von dieser Basisklasse aufgerufen,
  // wenn sich etwas an den Daten der Klasse (z.B.
  // an Attributen) ändert, das aber höchstens
  // einmal pro Render-Zyklus. DOM-Diffing patcht
  // UI-Updates nahtlos und schnell ein.
  @reactive()
  @connected()
  @debounce()
  #render() {
    if (this.render) {
      render(this.#root, this.render());
    }
  }
}

// Stellt den Funktionsumfang von Ornament dem Rest
// des Frameworks bereit
export * from "@sirpepe/ornament";
```

Eine Frontend-Komponente kann von dieser Basisklasse erben, und sobald wir die Komponenten-Klasse mit einem HTML-Tag assoziieren, übernimmt die HTML-Engine des Browsers den Rest. Das könnte für ein einfaches „Hallo Welt" wie in Listing 2 gezeigt aussehen.

Diese Klasse werfen wir in einen Browser, konfrontieren diesen dann mit dem HTML-Tag *<say-hello>* (optional mit dem *name*-Attribut), und fertig ist eine Web Component … jedenfalls fast. Denn die Decorators-Syntax mit dem @-Zeichen ist für Browser Mitte 2025 noch Terra incognita, auch wenn sie kommender Bestandteil des ECMAScript-Standards ist. Wir kommen also doch nicht ganz um etwas Tooling herum, doch das muss kein Problem sein – wenn wir es clever anstellen!

Die dritte Entscheidung für dieses Framework ist nämlich, Developer Experience zugunsten von allem anderen dezent zu depriorisieren, wenn auch nicht komplett aufzugeben. Die Marketing-Strategie der gängigen SPA-Frameworks hat seit jeher die Entwicklerschaft im Fadenkreuz und preist den nur mit ihnen erreichbaren Komfortfaktor an. Tatsächlich hat sich aber die Webplattform mittlerweile so weit entwickelt, dass der maximale nativ gebotene Komfortfaktor als „gar nicht mal so schlecht" einzuordnen ist. Das einzig echte Problem ist, dass bisher nicht alle Browser die für „gar nicht mal so schlecht" benötigten Features an Bord haben. Mit den passenden Tools lässt sich aber genau diese Lücke schließen – und zwar so, dass wir stets auf dem Pfad der Webstandards verweilen. Wir sind mal an der Spitze der nativen Features und manchmal sogar weiter vorne als die Browser. Wir achten aber strikt darauf, den Pfad der Webstandards nie vollends zu verlassen, sodass jedes Investment in Tools eine Investition in Zukunft und kein fahrlässiges Gezocke auf das Überleben irgendwelcher Frameworks oder die Launen irgendwelcher Maintainer ist. Da die Decorators-Syntax mit dem @-Zeichen ein definitiv kommender Webstandard ist, der bereits in allen möglichen Tools außer Browsern implementiert ist, ist ihr Einsatz so unriskant wie nur möglich – selbst wenn wir als Enabler ein Tool wie esbuild [7] brauchen. Dieses sorgt für den Support von Decorators, unterstützt Hot Reloading sowie einen Dev-Server und entschärft ein paar kleinere Papercuts … aber nicht mehr! Wer Node.js installiert hat, kann sich all das mit einem einzigen CLI-Kommando ins Haus holen:

```
$ npx esbuild src/index.js --bundle --target=es2022
  --format=esm --outdir=dist --watch --servedir=.
```

Und das war es schon! Mit zwei mikroskopischen Libraries, einem generischen JavaScript-Compiler, der Entscheidung für Webstandards und einem lächerlichen Maß an Glue Code haben wir ein sehr brauchbares Feature-Set implementiert:

- Unsere „Toolchain" (mit exakt einem Tool darin) erlaubt die Verwendung der neuesten Standard-Features und versorgt uns mit DX-Grundbedürfnissen wie Hot Reloading.
- Mit Web Components haben wir ein Komponentenmodell, das (weil in HTML-Elementen resultierend) allen Entwickler:innen bekannt ist und mit jeder Software, vom PHP-CMS bis zum React-Frontend, kompatibel ist.

Listing 2: Hallo-Welt-Beispiel

```
// src/say-hello.js

import { BaseElement, define, attr, string } from
  "./framework.js";

// Die Komponemten-Basisklasse erbt von unserer
// Framework-Basisklasse und kann daher ihre
// Renderlogik bequem in einer render()-Methode
// definieren. Der Decorator @define() meldet die
// Klasse beim Browser als ein neues HTML-Element
// an.
@define("say-hello")
export class SayHello extends BaseElement {
  // Definiert ein HTML-Attribut namens "name",
  // das vom Typ "string" ist und "Anonymous" als
  // Standardwert verwendet.
  @attr(string()) accessor name = "Anonymous";

  // Definiert das von <say-hello> implementierte
  // UI. Änderungen an relevanten Daten wie etwa
  // dem "name"-Attribut lösen automatisch ein
  // UI-Update aus.
  render() {
    return this.html`<p>Hello ${this.name}</p>`;
  }
}
```

- Das Komponentenmodell verwendet Klassen, was die Frage, wo wir den Zustand der Komponenten hinterlegen, praktischerweise vollautomatisch beantwortet.
- DOM-Diffing und ein paar Decorators machen das Programmiermodell relativ deklarativ, relativ performant und relativ komfortabel.

Diese Liste ist eher kurz, aber sehr viel mehr als „React out of the box" braucht sie auch nicht zu bieten. Ähnlich wie bei React müsste auch unser Framework für komplexere Aufgaben allerlei Ergänzungen erhalten, doch das ist kein Problem – wir nehmen Webstandards, wo möglich, und an Standards orientierte Libraries, wo nötig. Mit diesem Standards-first-Ansatz haben wir aber bereits mehr geleistet, als nur eine so gerade eben adäquate Featureliste zusammenzustellen.

Quo vadis, DIY-Framework?

Featurelisten sind das eine. Nachdem wir aber die ersten 99 Prozent dieses Artikels damit verbracht haben, uns über die Instabilität und Aufgeblasenheit der gängigen SPA-Frameworks zu ereifern, lohnt sich ein Blick darauf, wie sich unser Werk unter diesen Gesichtspunkten schlägt. Was macht unser Mini-Framework alles nicht?

- Wir haben null Vendor-Lock-in. Das Build-Tool ist komplett austauschbar, da wir es allein als Mittel der Standard- ▶

Durchsetzung verwenden – keine exotischen Plug-ins oder Spracherweiterungen weit und breit! Wenn Browser eines Tages Decorators unterstützen, können wir das Tooling ersatzlos entfernen, und sollte der Maintainer dem Irrsinn der Welt anheimfallen, gibt es mehrere Eins-zu-eins-Ersatz-Optionen … denn wir brauchen nicht mehr als Webstandards und das absolute Minimum an Komfort-Features. Stringbasierte Templating-Systeme mit DOM-Diffing wie uhtml gibt es wie Sand am Meer, Eins-zu-eins-Ersatz und Ornament wurde von einem untalentierten Spinner (dem Autor dieses Artikels) innerhalb einer Woche unter dem Einfluss bewusstseinseintrübender Elemente (SARS-CoV-2) zusammengeschraubt. Bei Bedarf kann es durch richtige Software wie Lit [8] ersetzt oder von durchschnittlich begabten Entwickler:innen reproduziert oder geforkt und gewartet werden.

- Es gibt nichts zu lernen, was Kenner des Browsers nicht ohnehin schon auf dem Kasten haben. Web Components sind nichts weiter als HTML-Elemente, also etwas, das alle Webentwickler:innen wie ihre Westentaschen kennen. Und alle, die schon mal mit Absicht einen Code-Editor geöffnet haben, sind in der Lage, eine Klasse mit den basalsten aller OOP-Techniken herunterzuschreiben. Vom verkalkten Netscape-Boomer bis zum jungen Hüpfer: Wo ein Wille ist, kann die Deklaration *class A extends B* entstehen. Selbst LLMs, die lobotimisiertesten aller Junior-Devs, sind in der Lage, unfallfreie JavaScript-Klassen zu generieren!
- Mangels Vendor- oder Technik-Lock-in sind unsere Komponenten außerdem kompatibel zu allem, denn alles – auch das neueste SPA-Framework – muss am Ende des Tages Webstandards wie HTML sprechen. Sollte unser Projekt also eines Tages von der reinen Lehre abweichen und ein „richtiges" Frontend-Framework übergeholfen bekommen, werden unsere Komponenten nahtlos in das neue System integrierbar sein.
- Da wir fast alle signifikanten Aufgaben an den Browser selbst abwälzen, besteht unser Framework aus praktisch nichts. Es gibt ein wenig Glue Code, doch der hält nur Features zusammen, die uns die Plattform bietet. Die wenigen Dependencies sind austauschbar und erhalten wenige Updates, da bei der Entwicklung von Template-Parsers und API-Umbau-Decorators wenig schiefgehen kann und wenig Innovationsspielraum bleibt. Kein Code ist so gut wie kein Code – was nicht da ist, kann nicht veralten, nicht langsam sein, nicht Ressourcen für die Wartung verschlingen.

Unser selbst gestricktes SPA-Toolkit macht zwar nicht viel, doch das ist in Ordnung – was nicht ist, kann immer noch werden. Und im Zuge des Werdens kann das Framework sich weiterhin treu bleiben, denn anders als Features sind die Qualitäten des Frameworks etwas, das allein auf unseren Entscheidungen beruht. Indem wir uns auf Standards besinnen und Tooling und Dependencies nur installieren, wenn sie eine brauchbare Exit Strategy erlauben, fördern wir zutage, was heutzutage allzu oft unter Megabytes an JavaScript begraben liegt: dass die Webplattform eine stabile Grundlage für Content und UIs aller Arten ist. Sie zeigt Texte an, rendert komplexe UIs für Industrieprozesse und ist die Grundlage des Infotainment-Systems von Luxusautos. Sie leistet all das, obwohl Webentwickler:innen auf einem bestehenden, soliden Fundament eine fragile JavaScript-Lasagne nach der anderen auftürmen. Sie leistet das alles aber auch, wenn wir die Lasagne im Ofen lassen.

Und diese Grundidee kann problemlos am Leben erhalten werden, selbst wenn das Framework wächst und sich entwickelt. Jedes neue Feature und jede neue Anforderung kann unter der Maßgabe „Standards first" betrachtet und auf eine Weise umgesetzt werden, die in weniger Code, weniger Dependency-Updates und weniger Risiko mündet.

Der Normalzustand im Frontend ist heutzutage das glatte Gegenteil: Jedes Framework ist der neue große Entwurf, der alles neu und alles richtig macht, gleich wie viel JavaScript dafür aufgewendet werden muss und wie viel Instabilität damit einhergeht. Doch mit einer etwas anderen Herangehensweise kann die Entwicklung eines Webfrontends auch ganz anders laufen. Wer einen Use Case hat, der nicht eine hyperinteraktive Frontend-App ist, benötigt vielleicht gar kein SPA-Framework, sondern schnell ladendes HTML plus ein bisschen JavaScript-Würze. Wenn besondere Anforderungen an Performance oder Stabilität gestellt werden, kann man sich, wie gesehen, mit wenig Aufwand einen Framework-Maßanzug schneidern.

Nur wer gedenkt, genau das Gleiche wie der Rest der Welt zu machen, und an den gleichen Problemen wie der Rest der Welt scheitern möchte, ist mit den gleichen Lösungen, wie sie der Rest der Welt verwendet, gut bedient. Aber wie wäre es mal mit einem Projekt, das schnell lädt, einmal pro Woche statt einmal pro Stunde ein Dependency-Update benötigt und das auch nach Jahren noch aus seinem Quellcode lauffähig gemacht werden kann? Der Framework-Markt hat hierfür kein Angebot. Also machen wir uns einfach, wie wir gesehen haben, unser eigenes. ■

[1] State of JavaScript 2024, https://2024.stateofjs.com/en-US
[2] HTMX, https://htmx.org
[3] Hotwired, https://hotwired.dev
[4] View Transitions, www.dotnetpro.de/SL2504-05SPADIY1
[5] uhtml auf GitHub, https://github.com/WebReflection/uhtml
[6] Ornament auf GitHub, https://github.com/SirPepe/ornament
[7] esbuild auf GitHub, https://esbuild.github.io
[8] Lit, https://lit.dev

Peter Kröner
war selbstständiger Webdesigner und –entwickler, bevor er 2010 Trainer und Berater für Webtechnologien wurde. Seither bereist er das Land im Namen von JavaScript, HTML5, React und TypeScript, über die er auch regelmäßig bloggt und podcastet.

dnpCode A2504-05SPADIY