Sonderdruck aus JavaSPEKTRUM

Architektur für Frontends

Beliebte Patterns und zu welchen Frontends sie passen

Während im Backend leidenschaftlich über Microservices, Self-Contained Systems oder Monolithen gestritten wird, scheint die Frontend-Welt nur noch aus Single-Page Applications zu bestehen. Doch wann hat „One Size Fits All" jemals in der IT funktioniert? Dieser Beitrag zeigt Alternativen und gibt Empfehlungen, wann welche Frontend-Architektur am besten passt.

Immer mehr Anwendungen verlagern sich vom klassischen Desktop ins Internet. Auch bei Apps geht der Trend mit Progressive Web Apps (PWA) gerade wieder mehr in Richtung Browser. Hier gibt es neue Möglichkeiten zur Aufteilung der Anwendung. Nachfolgend möchten wir die aktuell am meisten diskutierten Frontend-Architekturen vorstellen.

Aber warum müssen wir uns überhaupt um unsere Frontends Gedanken machen? Die „richtige" Architektur hängt immer von den Anforderungen und Rahmenbedingungen ab, denen eine Anwendung genügen soll. Hier sind die Benutzer deutlich anspruchsvoller als zu Zeiten rein statischer Webseiten geworden: Inhalte müssen auf jeder Art von Geräten (vom Desktop-Bildschirm bis zur Armbanduhr) ansprechend dargestellt, interaktiv und aktuell sein, und das auch, wenn gerade kein Internetzugang verfügbar ist. Zudem verdichten Frontends Informationen aus einer Vielzahl von Informationssystemen, die nicht immer unter Kontrolle eines einzelnen Unternehmens stehen.

Das alles treibt die Komplexität schnell in Höhen, die kein einzelnes Entwicklungsteam mit *dem* Framework mehr bewältigen kann. Stattdessen sollen verschiedene Teams an kleineren Anwendungsteilen arbeiten und dabei die für sie passende Technologie wählen können – ohne dass die Benutzer auf ein einheitliches Bedienkonzept und Design verzichten müssen.

Frontend-Monolithen

Wer kennt ihn nicht, den großen Monolithen? Es ist das am häufigsten anzutreffende (und vielleicht auch ein wenig verpönte) Pattern: Hierbei wird das gesamte Frontend für alle Backends in einer einzigen Anwendung ausgeliefert (**siehe Abbildung 1**). Zwar gibt es diesen Architekturstil praktisch seit es Software selbst gibt, aber mächtige Single-Page-App-Frameworks wie Angular, React und Vue haben ihm in den letzten Jahren wieder zu mehr Popularität verholfen.

Das Frontend integriert hierbei mehrere Backend-Anwendungen. Das ist komfortabel für Benutzer, da für sie alles nahtlos und in einheitlichem Look-and-Feel präsentiert wird. Auch für Entwickler ist dieser Architekturstil gerade zu Anfang eines Projekts vorteilhaft, da der Code in einem Repository liegt, über einen einzigen Build-Prozess gebaut und ausgeliefert werden kann. Häufig anzutreffen ist dieses Muster in horizontal geschnittenen Teams, in denen Frontend und Backend von getrennten Entwicklern entwickelt werden.

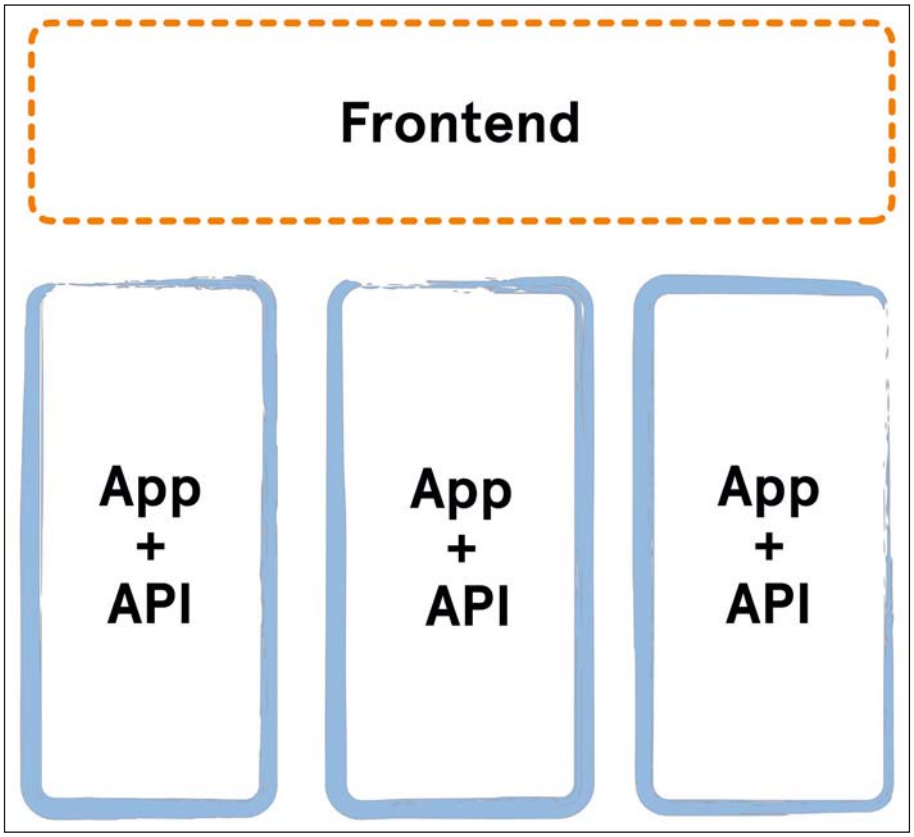

Abb. 1: Frontend-Monolith

Problematisch wird es, wenn mehrere Teams den Monolithen weiterentwickeln. Werden die einzelnen Backends von unterschiedlichen Teams bereitgestellt, ist auch jedes Mal ein Release des gesamten Frontends notwendig, was viel Koordinations- und Testaufwand erfordert. Auch die Größe des Frontends (in Lines of Code oder beteiligten Entwicklern) kann hier schnell zum Problem werden.

Abhilfe kann Modularisierung schaffen: Hierbei wird das Frontend in einzelne Module aufgeteilt, die von unterschiedlichen Teams weiterentwickelt und erst beim Deployment wieder zusammengefügt werden. Dieses Pattern ist auch als *Deployment-Monolith* bekannt.

Über ein Problem hilft aber auch Modularisierung nicht hinweg: Verwendet das Frontend ein bestimmtes Framework, so sind meist alle Module darauf festgelegt – im schlimmsten Fall bis auf die letzte Stelle der Version genau. Für die technische Weiterentwicklung ist das ein großes Hindernis, und Migrationen werden schnell zum Mammut-Akt.

Gerade für kleinere Projekte mit wenigen beteiligten Entwicklern eignet sich der Monolith sehr gut, da er den Integrationsaufwand einer verteilten Anwendung vermeidet. Für einige Anwendungen kann der Monolith auch technisch der einzig gangbare Weg sein, zum Beispiel für Mobile-Apps auf Android und iOS. Doch mit wachsender Projektgröße können der Koordinationsaufwand und die Größe der Codebasis zum Problem werden. Hier können Self-Contained Systems eine Alternative sein.

Self-Contained Systems

Die naheliegende Alternative zu einer einzigen, großen Anwendung ist die Aufteilung in mehrere unabhängige Teile. Ein populärer Vertreter dieses Architekturstils sind Self-Contained Systems ([SCS], **siehe Abbildung 2**). Die zentrale Idee dieses Ansatzes ist, dass eine Anwendung für eine fachlich abgeschlossene Domäne alles Notwendige selbst bereitstellt – von der Datenbank zur Business-Logik bis hin zum Frontend. Ein System wird dabei immer von genau einem Team betreut.

Die Frontends sind bei diesem Pattern von zentraler Bedeutung, da es ihre Aufgabe ist, aus den einzelnen getrennten Anwendungen für den Benutzer wieder ein integriertes System zu machen. Im einfachsten Fall geht das über simple Links von einer Applikation zur anderen. Da die einzelnen Anwendungsteile nur noch lose gekoppelt sind und komplett unterschiedliche Technologien und Release-Zyklen haben können, wird es schwerer, ein einheitliches Look-and-Feel für die Benutzer zu erreichen. Hier können Living-Styleguides [SGR] oder Komponentenbibliotheken Abhilfe schaffen.

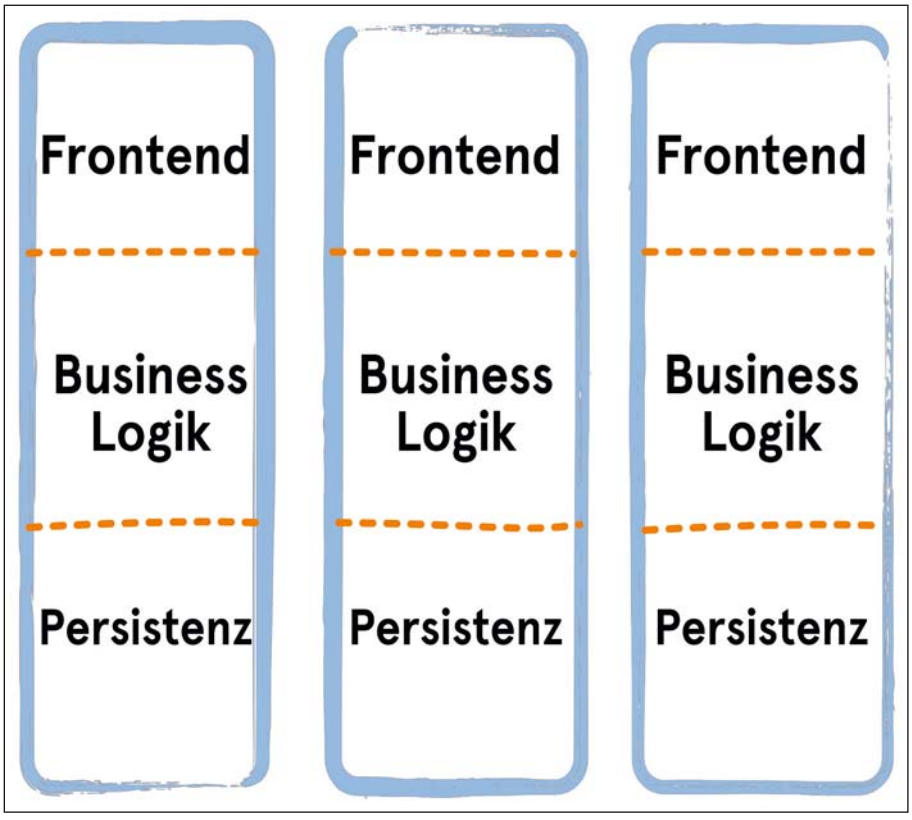

Abb. 2: Self-Contained Systems

Vorteile	Nachteile
schnell aufzusetzen	Koordinationsaufwand in großen Teams
einfache Integration in Continous-Integration und -Delivery	große Codebasis wird schnell unübersichtlich
eine Codebasis	Fehler wirken sich auf alle Teile der Anwendung aus

Tabelle 1: Vor- und Nachteile der Monolithen

Vorteile	Nachteile
Anwendungsteile können getrennt voneinander entwickelt und released werden	einheitliches Look-and-Feel schwieriger zu gewährleisten
freie Technologiewahl innerhalb der Systeme	Integration von Elementen unterschiedlicher Systeme auf einer Seite aufwendig
Ausfälle eines einzelnen Teils wirken sich nicht auf das Gesamtsystem aus	eventuell Redundanzen und Code-Duplikationen notwendig

Tabelle 2: Vor- und Nachteile der SCS

Vorteile	Nachteile
flexible Integration durch Komponenten	einheitliches Look-and-Feel schwieriger zu gewährleisten
bei rein clientseitiger Integration kein Server benötigt	SEO ohne serverseitiges Rendering schwierig
durch nachgeladene Komponenten kürzere Ladezeiten	

Tabelle 3: Vor- und Nachteile der Micro-Frontends

Gerade Single-Page Applications tun sich mit diesem Ansatz schwer: Hier ist das Frontend streng genommen ein eigenes System, was sich viel Geschäftslogik mit dem Backend teilt – was in SCS nicht erlaubt ist. Daher sind Technologien mit serverseitigem Rendering hierfür oft die bessere Wahl. Clientseitiges JavaScript sollte vor allem dazu verwendet werden, die Seiten um zusätzliche interaktive Elemente zu erweitern.
Dafür kann jedes System für sich technologisch aus dem Vollen schöpfen und die verwendeten Technologien und Frameworks frei für sich wählen.

Integration von Self-Contained Systems

Die besondere Herausforderung bei SCS besteht darin, die einzelnen Anwendungsteile für den Benutzer wieder zu einem zusammenhängenden Gesamtsystem mit einheitlichem Look-and-Feel zusammenzuführen.
Eine einfache Möglichkeit hierfür ist, die einzelnen Frontends zu verlinken. In einem Webshop könnten zum Beispiel die Produktübersicht und der Bestellprozess in unterschiedlichen Systemen abgebildet werden. Mit einem Klick auf den Warenkorb wechselt der Benutzer die Anwendung und kann beispielsweise über einen Link auf dem Artikelnamen im Warenkorb zurücknavigieren.
Die Daten können dabei wahlweise auf der Clientseite oder im Backend übergeben werden. Im Frontend lässt sich dies beispielsweise über Parameter in der URL, Cookies oder den Local-Storage des Browsers lösen.
Doch ist die Trennung zwischen den Anwendungsteilen nicht immer so eindeutig. Häufig teilen sich Frontends auch gemeinsame Elemente, wie ein Menü zur Navigation oder einen Footer mit Links und rechtlich geforderten Angaben. Liefert jeder Teil seine eigene Version dieser Elemente, wird es schwierig, ein einheitliches Look-and-Feel zu gewährleisten. Shared Code, wie eine Library mit der geteilten Komponente, die alle Frontends einbinden, hat wiederum den Nachteil, dass erneut Annahmen über die verwendeten Frameworks getroffen werden müssen. Hier können Server-Side-Includes helfen.
Server-Side-Includes (SSI) sind die „moderne“ Variante von iFrames: Sie erlauben es, über HTML-Mark-up Inhalte zur Laufzeit in die eigene Seite einzubinden (**siehe Listing 1**). Hierfür parst zum Beispiel ein Webserver, Reverse-Proxy oder Cache das HTML vor dem Ausliefern an den Client und fügt die referenzierten Seiten in den DOM ein [Wiki]. Im Gegensatz zu iFrames ist diese Form der Transklusion für den Browser transparent, wodurch auch alle Stylesheets auf den eingebundenen Inhalt angewendet werden und es keine Einschränkungen für JavaScript beim Zugriff auf diesen Teil des DOMs gibt. Da die Seite vom Server jedoch erst ausgeliefert werden kann, wenn alle Teile geladen wurden, kann sich das negativ auf die Ladezeit auswirken. Dem lässt sich durch entsprechendes Caching der ohnehin meist statischen Komponenten vorbeugen.
Neben SSI gibt es noch mindestens einen weiteren Standard mit ähnlichen Design-Zielen: Edge-Side-Includes (ESI, [W3C-a]). Im Gegensatz zu SSI wird hier die Seite nicht schon auf dem Webserver oder Reverse-Proxy zusammengesetzt, sondern von einem dedizierten Cache (wie Varnish) oder einem Content-Delivery-Network (CDN). Hierfür ist somit zusätzliche Infrastruktur notwendig. Dafür bietet ESI deutlich mehr Möglichkeiten für das Caching und die Lastverteilung: Statische Inhalte kommen aus dem Cache, und das eigentliche System kann sich um den dynamischen Teil der Seite kümmern. Durch

```
<html>
  <body>
    <!--# include file="header.html" -->
  </body>
</html>
```

Listing 1: Server-Side-Includes

geschicktes Caching können sogar Downtimes des eigentlichen Systems kurzzeitig überbrückt werden.
Als Alternative zu serverseitigen Technologien gibt es auch clientseitige Lösungen – die einfachste kennen Sie sicher noch als „Ajax“. Ein Architekturstil, der auch komplexe clientseitige Frontends zu entwickeln hilft, ist Micro-Frontends.

Micro-Frontends

Micro-Frontends verfolgen ähnliche Ansätze und Ziele wie Self-Contained Systems. Statt jedoch ganze Seiten über Hyper-Links zu verbinden, stellen Teams zusätzliche Komponenten bereit, die andere einbinden können [MFE].
Im vorhin skizzierten Webshop ist der Warenkorb als Hyperlink in die Produktseite integriert. Soll nun zusätzlich noch die Anzahl der Artikel im Warenkorb im Icon angezeigt werden, dann könnte es hierfür zwei technische Umsetzungen geben. In der ersten, nicht Micro-Frontend konformen Umsetzung wird die zusätzliche Logik in der Produktseite integriert. Dies hat den Nachteil, dass nun die beiden Systeme miteinander kommunizieren müssen.
Besser im Sinne des Micro-Frontends-Ansatzes ist es, eine Komponente für den Warenkorb-Link zu erstellen, die vom Produktteam eingebunden werden kann. Ähnliches kennt man schon länger von Embedded YouTube-Videos oder Google-Maps Widgets.
Der Unterschied zu SCS ist der klare Fokus auf komponentengetriebene Entwicklung. Mit Web Components [W3C-b], ein Webstandard des W3C, können eigene HTML-Elemente erstellt werden, die ähnlich wie Widgets aufgebaut sind [Tre16]. Innerhalb von Web Components werden Layout (HTML), Styling (CSS) und Logik (JavaScript) gekapselt und von der eigentlichen Verwendung im DOM getrennt. Somit verhalten sie sich für den Verwender wie native HTML-Elemente (**Listings 2 und 3**). Hierfür nutzen sie den Shadow DOM, der einen eigenen Scope unabhängig vom DOM der Hauptseite eröffnet.
So können auch unterschiedliche Framework-Versionen auf einer Seite existieren.

```
class MyWebComponent extends HTMLElement {
  constructor() {
    super();
    this.innerHTML = '<h1>MyWebComponent</h1>';
  }
  disconnectedCallback() { ... }
}
window.customElements.define('my-web-component', MyWebComponent);
```

Listing 2: Web Components

```
<html>
  { ... }
  <body>
    <my-web-component></my-web-component>
  </body>
</html>
```

Listing 3: Verwendung von Web Components

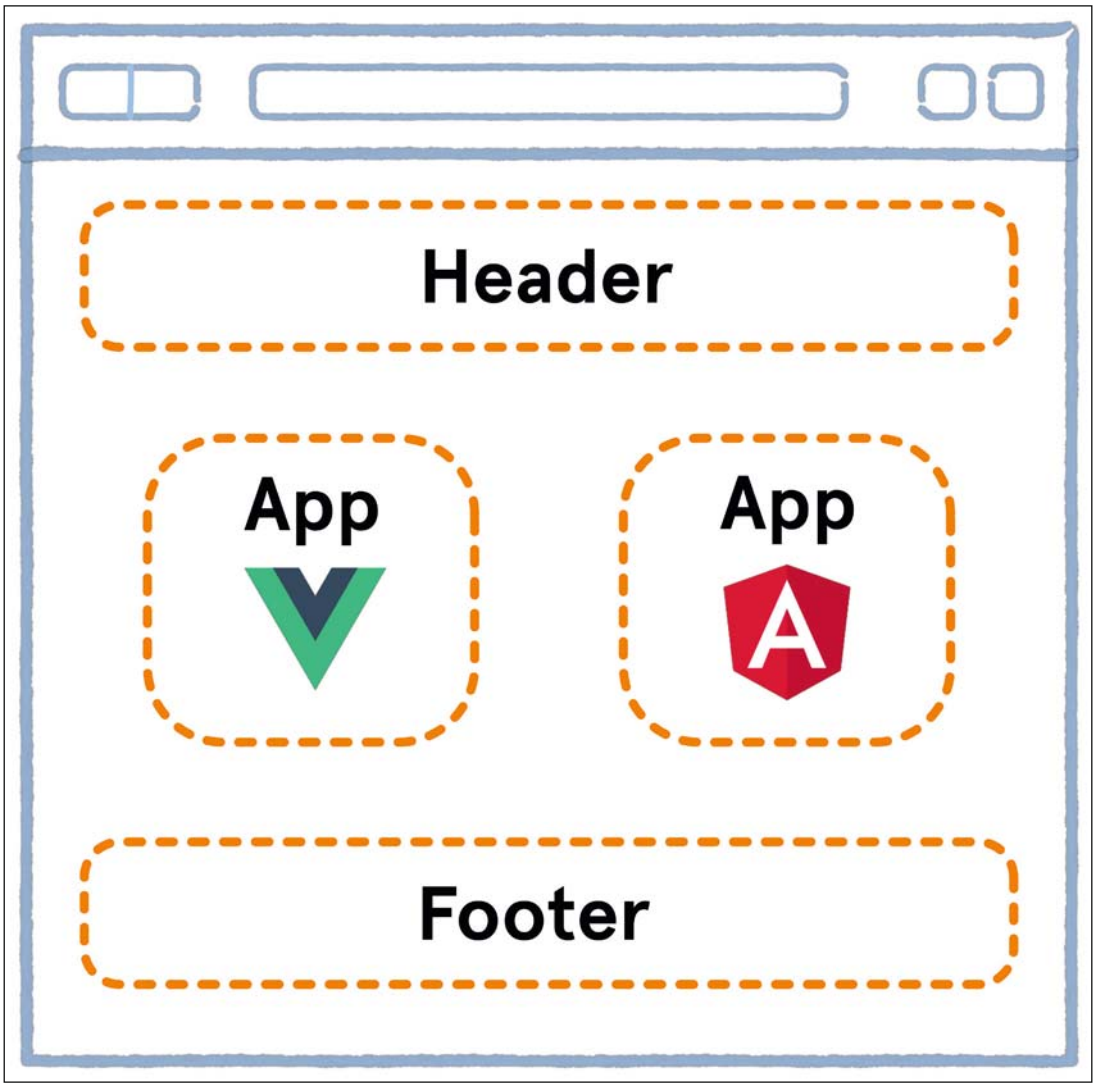

Abb. 3: Micro-Frontend

Literatur & Links

[MFE] M. Geers, Micro Frontends, siehe: siehe: **https://micro-frontends.org/**

[SCS] Self-Contained Systems, siehe: **https://scs-architecture.org/**

[SGR] Website Style Guide Resources, siehe: **http://styleguides.io/**

[Tre16] T. Tretau, Komponentenbasierte Softwareentwicklung für das Web mit dem Web Component-Standard, in: OBJEKTspektrum, 06/2016

[W3C-a] Edge Side Includes, W3C, siehe: **https://www.w3.org/TR/esi-lang**

[W3C-b] Web Components Current Status, W3C, siehe: **https://www.w3.org/standards/techs/components**

[Wiki] Wikipedia, siehe: **https://en.wikipedia.org/wiki/Server_Side_Includes**

Fast alle populären Web-Frameworks und Browser unterstützen Web Components und damit auch die Koexistenz verschiedener Web-Frameworks auf einer Webseite (**siehe Abbildung 3**).
Oft sollen die einzelnen Web Components zwar unabhängig voneinander entwickelt werden, zur Laufzeit jedoch trotzdem miteinander kommunizieren. In einem Webshop soll sich zum Beispiel die Anzahl von Waren erhöhen, wenn der Kunde ein Produkt zum Warenkorb hinzufügt. Das sollte dann über klar definierte Schnittstellen erfolgen. Damit dies auch zwischen unterschiedlichen Frameworks funktioniert, bieten sich Standard-Mechanismen wie JavaScript-Events oder das einfache Ändern von Attributen auf HTML-Elementen an.
Während SCS serverseitiges Rendering stark bevorzugen, ist dies bei Micro-Frontends nicht der Fall. Bei Micro-Frontends ist es möglich, dass Frontend und Backend getrennte Anwendungen sind, die nur über eine definierte Schnittstelle miteinander kommunizieren. Dies kann Vorteile haben, insbesondere wenn sich das Frontend schneller weiterentwickelt als das Backend oder mehrere Frontends, zum Beispiel Mobile und Desktop, für ein und dasselbe Backend entwickelt werden. Der Nachteil ist, dass dafür unter Umständen Business-Logik dupliziert werden muss.
Micro-Frontends haben im Wesentlichen dieselben Vor- und Nachteile wie SCS: Die verschiedenen Komponenten können mit verschiedenen Technologien von einzelnen Teams getrennt voneinander entwickelt werden. Zudem können die Komponenten getrennt voneinander deployed werden, da sie zumeist erst während der Laufzeit miteinander integriert werden. Ausfälle von einem Frontend wirken sich dadurch nicht auf die Verfügbarkeit der gesamten Anwendung aus.
Eine Herausforderung ist dabei, ein einheitliches Look-and-Feel zu gewährleisten, ohne zu viel Code zu teilen. Wenn die einzelnen Micro-Frontends komplett auf Browser zusammengebaut werden, kann das Nachteile bei der Suchmaschinenoptimierung (SEO) haben, da die Seite für die Suchmaschine schwerer zu indizieren ist.

Wann eignet sich welche Architektur?

Für kleinere Projekte, an denen wenige Entwickler arbeiten, sind Monolithen oft der richtige Ansatz. Sie bringen wenig Overhead mit sich und vermeiden die Komplexität verteilter Systeme und viele der Probleme, denen dieser Ansatz seinen manchmal schlechten Ruf verdankt, lassen sich über saubere Modularisierung vermeiden. Bei Apps führt zudem meist schon aufgrund der Auslieferung über den App-Store kein Weg um einen Monolithen herum.
Bei großen Projekten mit hohen Anforderungen an Verfügbarkeit und vielen beteiligten Teams eigenen sich insbesondere Self-Contained Systems. Sie erlauben den höchsten Grad an Unabhängigkeit zwischen den Teams.
Micro-Frontends verfolgen einen ähnlichen Ansatz mit deutlichem Schwerpunkt auf clientseitigen Technologien. Das hat Vorteile für besonders interaktive Frontends, wo technologisch aus dem Vollen geschöpft und alle Möglichkeiten des Browsers ausgereizt werden können.

Fazit

Die Praxis ist aber nicht schwarz-weiß – nirgends steht, dass ein SCS nicht auch Komponenten anderer Systeme einbinden kann, solange es seine Aufgabe trotzdem noch selbstständig erfüllen kann. Auch Micro-Frontends vertragen sich prima mit serverseitigem Rendering, um Progressive Enhancement zu bieten oder SEO in den Griff zu bekommen. Und selbst in die „monolithischste“ App lassen sich fremde Inhalte per Web-View integrieren. ||

Die Autoren

Gregor Tudan
(gregor.tudan@cofinpro.de)
arbeitet bei Cofinpro als Entwickler und Architekt an einer großen Online-Plattform für mehrere Hundert Banken. Daher liegt sein Schwerpunkt auf Mandantenfähigkeit und Continuous-Integration.

Kevin Kessenich
(kevin.kessenich@cofinpro.de)
ist langjähriger Verfechter von Clean Code und begeisterter Webentwickler.