architektur SPICKER

Übersichten für die konzeptionelle Seite der Softwareentwicklung

MEHR WISSEN IN KOMPAKTER FORM:
Weitere Architektur-Spicker gibt es als kostenfreies PDF unter www.architektur-spicker.de

NR. 9

Moderne Frontend-Architektur

Single-Page Applications (SPAs) verschieben den Schwerpunkt einer Anwendung in Richtung Browser. Wie gehen Sie mit den daraus resultierenden architektonischen Herausforderungen um?

IN DIESER AUSGABE

Worum geht's?

- Ihre Frontend-Architektur ist veraltet. Wie finden Sie eine passende neue Web-Frontend-Architektur?
- Ihre Anwendung muss mehr als 10 Jahre leben. SPA-Frameworks sind aber für ihre Kurzlebigkeit verrufen, wie gehen Sie damit um?
- Ihre Anwendung ist groß und nicht mit einem Team zu entwickeln. Welche Kompromisse müssen Sie treffen?
- Wie modularisieren Sie und wie fügen Sie die Anwendung wieder zu einem konsistenten Ganzen zusammen?

Typische Anforderungen für SPAs

Single-Page Applications (SPAs) erlauben hochinteraktive Web-Anwendungen, indem sie auf Benutzereingaben durch Veränderung einer einzigen HTML-Seite reagieren.

Im Folgenden finden Sie typische Anforderungen für SPAs nach unterschiedlichen Blickwinkeln kategorisiert. Ihre Anwendung hat eine Teilmenge davon. ☒ Kreuzen Sie die wichtigsten Anforderungen an!

User Experience

Eine Software, die keiner bedienen kann, ist wertlos.

- ◯ Konsistente Darstellung
- ◯ Antwort-Verhalten
  - Erste Ansicht bei Aufruf
  - Verzögerung bei Interaktionen
- ◯ Darstellung einer Vorschau (z.B. für soziale Medien)
- ◯ Informationsgehalt
  - nicht zu viel
  - nicht zu wenig

Developer Experience

Auch Entwickler sind Stakeholder und wollen effektiv entwickeln

- ◯ Autonome Teams
- ◯ Geringe kognitive Last bei Konzepten
- ◯ Spannende Technologien
- ◯ Dokumentation / Stack Overflow / Community
- ◯ Framework passend zur Erfahrung, fühlt es sich gut an?
- ◯ Programmier-Stil (z.B. objekt-orientiert, funktional, deklarativ)

Management, Produktverantwortung

Software muss effizient entwickelt und auch irgendwann ausgeliefert werden.

- ◯ Markt für Entwickler
- ◯ Effiziente Entwicklung
- ◯ Time to Market
- ◯ Schrittweises Ausrollen
- ◯ Story für Migration
- ◯ Skalierbarer Betrieb
- ◯ Lange Lebensdauer

System-Architektur

Lösungen sollen zum Problem passend und zukunftsfähig sein.

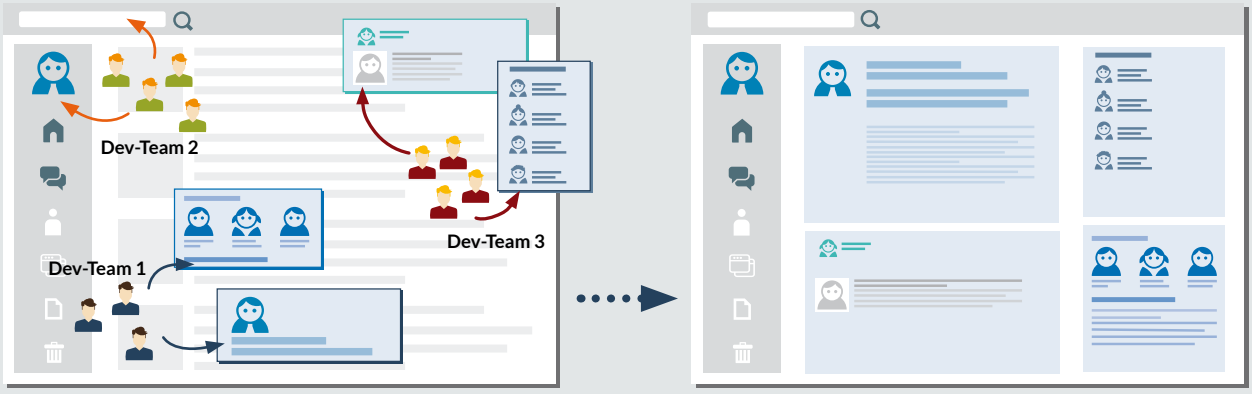
- ◯ Skalierbare Entwicklung
- ◯ Kein Over- oder Under-Engineering
- ◯ Wartbarkeit
- ◯ Viele Freiheitsgrade
- ◯ Einheitlichkeit / klare architektonische Richtlinien
- ◯ Wiederverwendung von Komponenten und Konzepten
- ◯ Externe Abhängigkeiten (möglichst minimal und stabil)

Äußere Architektur

Wie zerteilen Sie eine große Anwendung in sinnvolle Anwendungsteile und integrieren diese wieder?

Zerlegung und Zusammenspiel

Große Anwendungen erfordern es, dass mehrere Teams diese gemeinsam entwickeln.
Die Anwendungsteile sollen sich zu einer möglichst schlüssigen Anwendung zusammensetzen lassen.



Formel Zusammensetzen = Integration der Teile + Kommunikation zwischen Teilen

Optionen für die äußere Architektur

Wie können Sie Ihre Anwendung strukturieren? Die folgende Tabelle zeigt Optionen auf:

Option	Was sind Anwendungsteile?	Wann und wo erfolgt die Integration der Teile?	Wie kommunizieren die Teile?	Wie unabhängig kann deployed werden?
Modularer Monolith Eine einzelne Anwendung aus Modulen, die beim Build integriert wird	Module, alle mit einem einzigen Framework umgesetzt, z.B. • npm package • Angular Module	Beim Bauen der Anwendung	• Zentraler Zustand, z.B. Redux • Events	Nur zusammen als Monolith
Micro Components Auf einer einzelnen Webseite werden unterschiedliche Teile per iFrame oder Web Component gleichzeitig dargestellt	Eigenständige Web-Applikationen, jede mit beliebiger Technik umgesetzt	Zur Laufzeit auf einer Seite im Browser	• Events • Cookies (z.B. Auth Token) • LocalStorage	Jede Web-Applikation einzeln und unabhängig von den anderen
Klassische Links Jeder Teil nimmt die komplette Seite ein. Andere Anwendungsteile werden per Link verbunden.	Eigenständige Web-Applikationen, jede mit beliebiger Technik umgesetzt	Mehrere Seiten oder Tabs im Browser. Mehrere Tabs können gleichzeitig offen sein	• URL-Parameter (bei Links zwischen Anwendungsteilen) • Cookies • LocalStorage	Jede Web-Applikation einzeln und unabhängig von den anderen

Empfehlung: **So lange die Anwendung überschaubar bleibt und in einem Team entwickelt werden kann „Modularer Monolith“. Dann Kombination mit „Micro Components“ wenn viele Teile gleichzeitig dargestellt werden sollen oder andernfalls „Klassische Links“.**

Auswahl des Komponenten-Frameworks

Kurzvorstellung

Im Folgenden sehen Sie Steckbriefe der gängigen Frameworks für die Entwicklung von SPAs.

Angular

Motto: do it the Angular way

- Das Framework von Google
- Inkompatibler Nachfolger von Angular.js
- Name für Version 1.x „Angular.js“, ab Version 2 „Angular“
- Klare Ideen und Vorgaben
- Observables als durchgängiges Muster (Reactive Extensions for JavaScript, rxjs)
- Komponenten als Klassen
- TypeScript Sprache der Wahl

React

Motto: JavaScript is king

- Das Framework von Facebook (Instagram und Facebook sind damit gebaut)
- Kein komplettes Framework, nur reaktive Rendering-Bibliothek
- Templates sind JavaScript mit Spracherweiterung JSX
- Kein Komponenten-Stil vorgeschrieben
  - Ursprünglich Klassen
  - Jetzt Funktionen mit „Hooks“ für Zustand, Lifecycle, Seiteneffekte, etc.

Vue (sprich „View“)

Motto: HTML is king

- Inoffizieller Community-Nachfolger von Angular.js
- Mit minimalem Aufwand einsetzbar
- Simple Konzepte
- Komponenten als Klassen möglich
- Bei Bedarf komplettes Framework
- Version 3 komplett in Type-Script

Web Components

Motto: Anti-Framework

- Kein Framework, nur Standard Browser-API
- Definition eigener Tags als Klassen
- Isolation von HTML / CSS in Komponenten
- Komplette Anwendungsentwicklung nur mit Bibliotheken zum reaktiven Rendering zumutbar (z.B.: lit-element als Nachfolger von Polymer). In den Skalen unten als „+lib“ eingetragen.

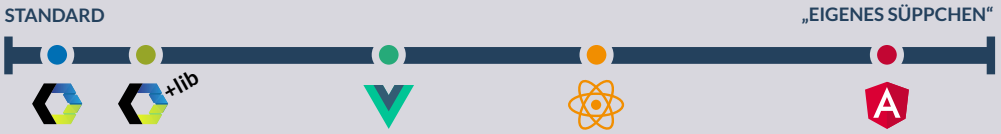
Framework-Vergleich

Grundsätzlich folgen alle Frameworks einem komponentenbasierten Ansatz und eignen sich für alle Aufgaben. Häufig zählen bei der Framework-Auswahl daher der Geschmack oder die Vorkenntnisse der Entwickler.

In Bezug auf manche Aspekte stellen sich die Frameworks jedoch unterschiedlich dar. Die folgenden Skalen geben für wichtige Unterschiede unsere Einschätzung wieder und bieten eine grobe Orientierung.

Seit wann ist das Framework seriös nutzbar?

Wie nah ist das Framework an Browser- und Web-Standards?



Wie stark leitet das Framework Sie bei der Entwicklung (opinionated)?

Wie stark müssen Sie auf Weiterentwicklungen im Framework reagieren?

Wie viel bindet man sich mit einem Framework ans Bein? Kann man das Framework im Notfall noch selbst warten?

Wie vollständig ist das Framework für ein typisches großes Vorhaben?

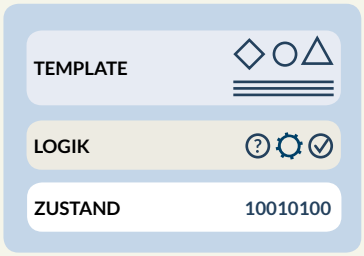
Innere Architektur

Komponenten als Bausteine

Wie setzen sich Anwendungsteile aus Komponenten zusammen und wie kommunizieren diese miteinander?

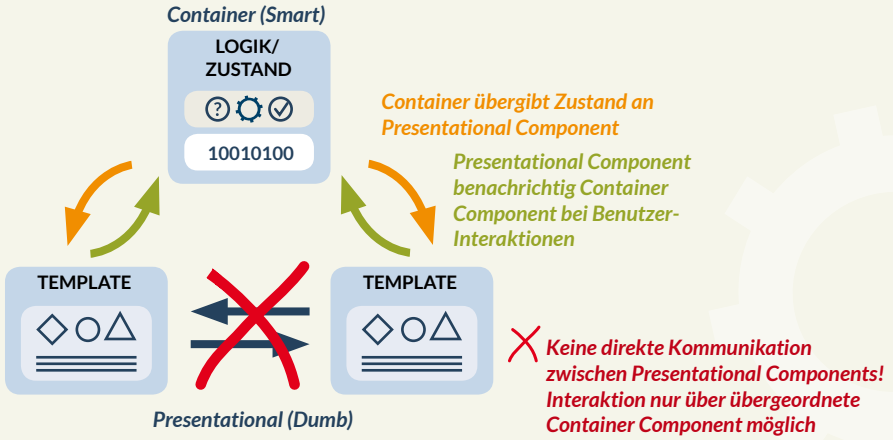
Es hat sich als Muster etabliert, Anwendungsteile aus **zwei Arten von Komponenten** zusammen zu setzen. Die **Container Components** kümmern sich um Logik und können wiederum andere Container und auch Presentational Components enthalten. **Presentational Components** kümmern sich um die Darstellung und enthalten keine nennenswerte Logik. Sie können ebenfalls andere Container und auch wieder Presentational Components enthalten.

Komponente



In allen Frameworks setzen sich die Anwendungsteile aus Komponenten zusammen. Diese können Template, Logik und Zustand kapseln.

Klarer Daten- und Kontrollfluss zwischen den Komponenten



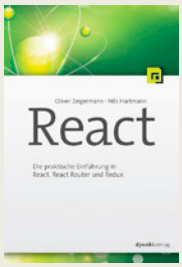
Komponenten-Bibliotheken enthalten vor allem wiederverwendbare Presentational Components. Container Components sind typischerweise fachspezifisch und daher nicht über den Anwendungsfall hinaus wiederverwendbar.

7 Typische, weiterführende Fragestellungen

Bei der Detaillierung der inneren Architektur tauchen immer wieder dieselben Fragestellungen auf. Die folgenden sollten Sie ihm Rahmen ihres Vorhabens beantworten.

1. Reicht das Muster von Container/Presentational Components oder brauche ich weitere Muster?
2. Wenn Komponenten nur über ihre Eltern kommunizieren können, wie vermeide ich das Hochwandern von allem Zustand und aller Logik in eine einzige „Gott-Komponente“?
3. Viele nutzen TypeScript als Programmiersprache. Sollte ich das auch machen oder geht auch JavaScript?
4. Sollte ich für die Kommunikation mit dem Backend GraphQL nutzen oder reicht nach wie vor REST?
5. Wie teste ich meine Anwendung automatisiert? Welche Teile sollte ich wie testen? Wie kann ich meine Anwendung schreiben, um Testen zu vereinfachen?
6. Wie trenne ich technischen Code, z.B. Backend-Calls oder Lifecycle-Methoden und Geschäftslogik voneinander und mache mich unabhängig vom Framework?
7. Nutze ich ein spezielles Framework für Zustandsmanagement (z.B. Redux) oder reichen die Bordmittel des Komponenten-Frameworks?

Weitere Informationen



Print

- O. Zeigermann, N. Hartmann: „React - Die praktische Einführung in React, React Router und Redux“, dpunkt-Verlag, https://reactbuch.de/

Online-Ressourcen

- Container und Presentational Components: https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0
- Redux Zustandsmanagement: https://redux.js.org/
- Bibliothek zum reaktiven Rendering von Web Components: https://lit-element.polymer-project.org/
- Reactive Extensions for JavaScript: https://rxjs-dev.firebaseapp.com/

Der Autor dieses Spickers

- Oliver Zeigermann
  Kontakt: oliver.zeigermann@embarc.de
  Twitter: @DJCordhose

Wir freuen uns auf Ihr Feedback: spicker@embarc.de

https://architektur-spicker.de

embarc Software Consulting GmbH
https://www.embarc.de
info@embarc.de

SIGS DATACOM
FACHINFORMATIONEN FÜR IT-PROFESSIONALS
https://www.sigs-datacom.de
info@sigs-datacom.de