

RICH CLIENT

WAS BEIM LETZTEN MAL GESCHAH

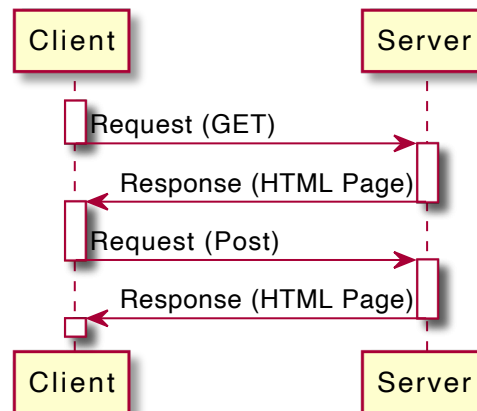
- Multi Page Application
- JSF
- Wiederholung MVC

IN DIESER VORLESUNG

- Single Page Applications (SPA)
 - Was ist eine SPA?
 - Was sind Vorteile?
 - Was sind Nachteile?
- Wie baue ich eine SPA? (Frontend Architekturen)
 - Component Architektur
 - Micro Frontends
- Vergleich zwischen SPA Frameworks (Angular, React und Vue)
- Praxis: Bau unserer Todo Anwendung in Angular

WIEDERHOLUNG

Wie funktioniert die Navigation bei Multi Page Anwendungen?



MOTIVATION

DATEN VOM BACKEND

```
1 <html>
2   <head>
3     <link rel="stylesheet" href="style.css">
4     <script>{javascript}</script>
5   </head>
6   <body>
7     <div>
8       // some data
9     </div>
10  </body>
11 </html>
```

STYLESHEETS

```
1 <html>
2   <head>
3     <link rel="stylesheet" href="style.css">
4     <script>{javascript}</script>
5   </head>
6   <body>
7     <div>
8       // some data
9     </div>
10  </body>
11 </html>
```

STYLESHEETS

- enthalten häufig ähnliche Informationen
- könnten einmalig ausgeliefert werden

```
1 <style>
2   label {
3     font-size: 12pt;
4     color: blue;
5   }
6
7   input {
8     font-size: 10pt;
9     color: green;
10    height: 10px;
11    width: 20px;
12  }
13 </style>
```


JAVASCRIPT

```
1 <html>
2   <head>
3     <link rel="stylesheet" href="style.css">
4     <script>{javascript}</script>
5   </head>
6   <body>
7     <div>
8       // some data
9     </div>
10  </body>
11 </html>
```

JAVASCRIPT

- ebenfalls repetitiv
- auf mehreren HTML Seiten braucht es gleiche Funktionalität

```
1 <script>
2   function openDropdown() {
3       // do it
4   }
5
6   function doSomeFancyAnimation() {
7       // do it
8   }
9 </script>
```

HTML STRUKTUR

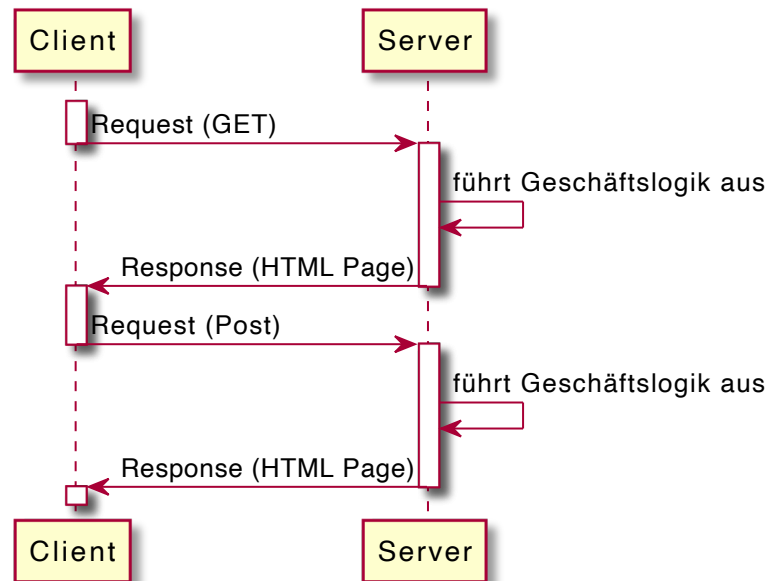
```
1 <html>
2   // head
3   <body>
4     <header></header>
5     <nav></nav>
6     <div>
7       // some data
8     </div>
9     <footer></footer>
10  </body>
11 </html>
```

HTML STRUKTUR

- dynamischer Anteil der Seite beschränkt sich auf Informationen

```
1 <html>
2   // head
3   <body>
4     <header></header>
5     <nav></nav>
6     <div>
7       // some data
8     </div>
9     <footer></footer>
10  </body>
11 </html>
```

WARTEZEITEN NACH DEN REQUESTS



Speaker notes

- User können in den Wartezeiten, bis die nächste Seite geladen wurde, nichts machen.
- Es wird außerdem kein Loadingspinner etc. angezeigt
- Bei SPA's wäre dies möglich

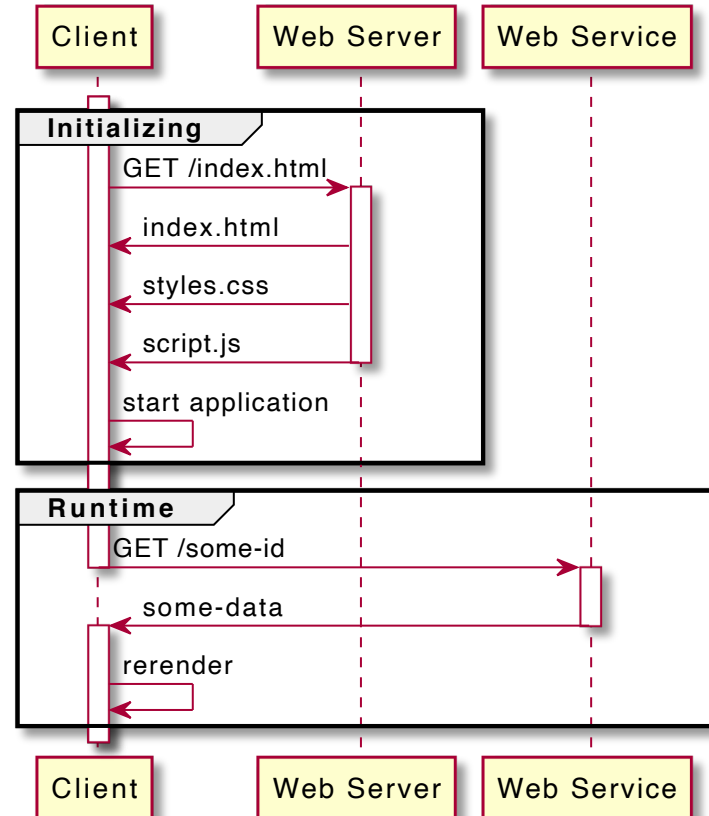
SINGLE PAGE APPLICATION

“A single-page application is exactly what its name implies: a JavaScript-driven web application that requires only a single page load.”

JavaScript - The Definitive Guide

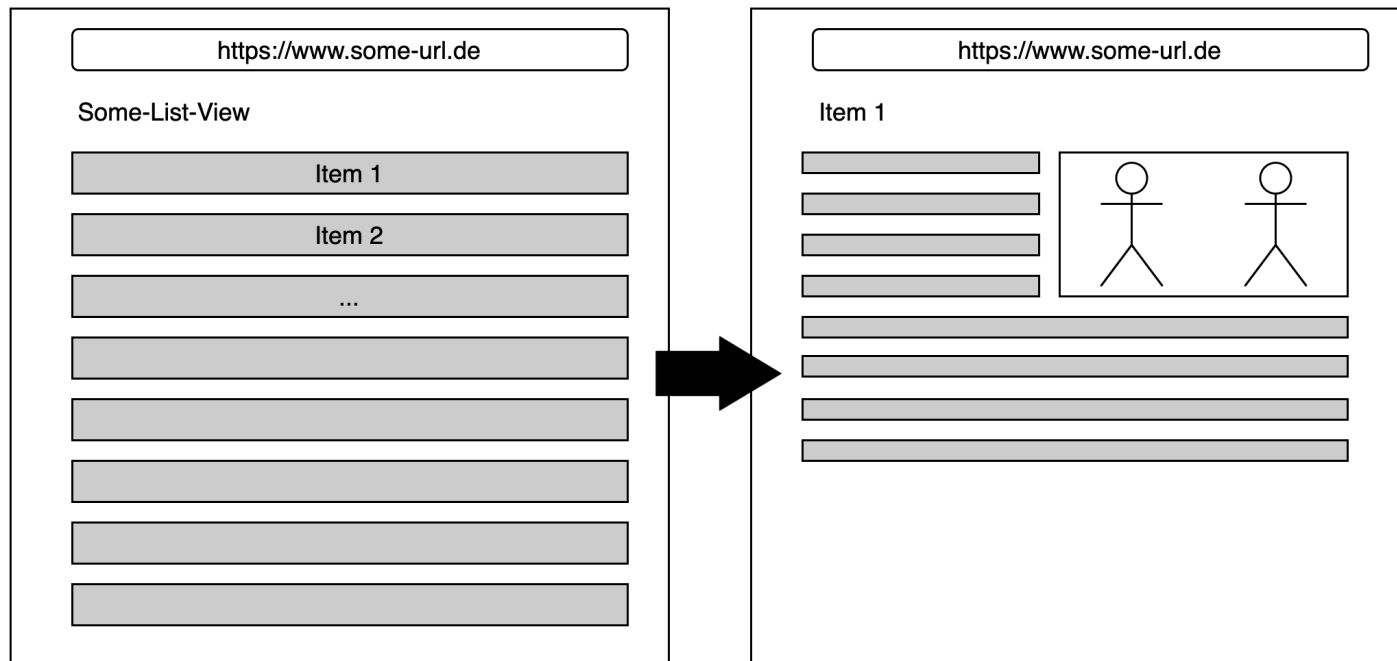
5th ed., O'Reilly, Sebastopol, CA, 2006

SINGLE PAGE APPLICATION KONZEPT



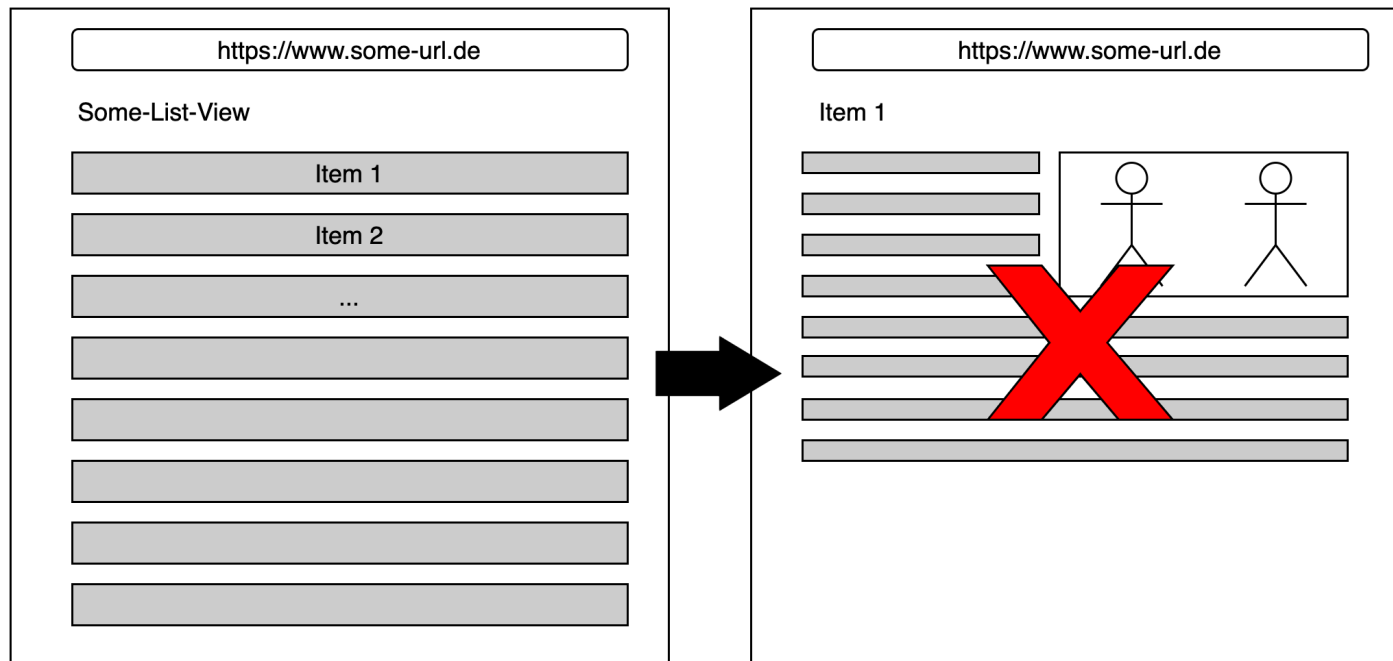
ROUTING?

- ist eigentlich nicht notwendig
- Anwendung macht einfach ein rerender



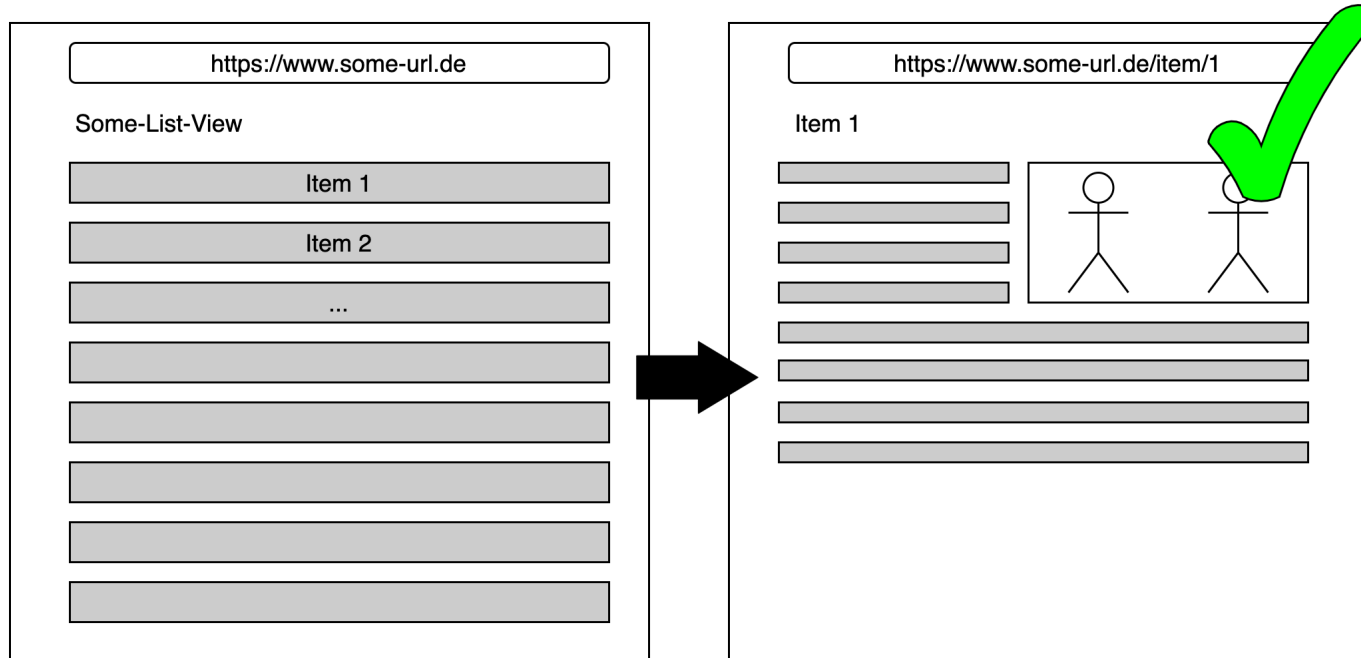
ALSO KEIN ROUTING?

- URL bleibt über die Laufzeit gleich
- teilen eines Links einer bestimmten Ressource?



ROUTING

- wir brauchen Routing in SPA's doch!
- es passiert ein pseudo Routing
- SPA Frameworks liefern Routing mit oder es gibt Libraries
- dazu später mehr ...



VORTEILE EINER SPA

- Reduktion der übertragenen Daten
- bessere User Experience
- weniger Serverressourcen
- Session Clientseitig (Server ist Stateless)
- Hybride Anwendung auch mobile einsetzbar

Speaker notes

- SPA's verhalten sich häufig wie App's. Daher können hybride Anwendungen auch als App eingesetzt werden.

REDUKTION DER ÜBERTRAGENEN DATEN

hier reden wir von Daten zur "Runtime"

```
1 <html>
2   // head
3   <body>
4     <header></header>
5     <nav></nav>
6     <div>
7       // some data
8     </div>
9     <footer></footer>
10  </body>
11 </html>
```

Speaker notes

- Erinnerung an das UML Initializing vs Runtime
- Erinnerung, dass nur noch Daten übertragen werden, keine ganzen HTML Seiten

BESSERE USER EXPERIENCE

- kürzere Response Time
- weniger BE Request notwendig

Speaker notes

- Kürzere Response Time
 - durch weniger Daten die übertragen werden müssen
- weniger BE Requests notwendig
 - Fehlermeldungen etc. können bereits ohne BE Requests angezeigt werden
 - Auch Geschäftslogik kann direkt im Frontend ausgeführt werden
- Seite ist während eines BE Requests benutzbar
 - Durch Loadingspinner etc. bekommt der Nutzer ein direktes Feedback auf seine Action
 - Nutzer fordert Daten an und kann sich dann mit etwas anderem beschäftigen, bis die Daten geladen sind.
 - In der Realität wird die Seite meistens nicht benutzt während eines BE Requests
- asynchrones Nachladen der Daten
 - der Nutzer kann bereits mit ersten Daten interagieren, während andere noch geladen werden.

WENIGER SERVERRESSOURCEN

- Rendering läuft auf dem Client
- Geschäftslogik kann auf dem Client laufen
 - weniger BE Requests notwendig
- Server kümmert sich nur um die Daten

VORTEILE EINER SPA

- Clientseitige Session
 - Server kann Stateless sein
 - Skalierbarkeit des Servers
 - Loadbalancing ist einfacher
- Hybride Anwendungen
 - SPA's sind ähnlich wie App's
 - React Native oder Flutter zeigen wie es geht

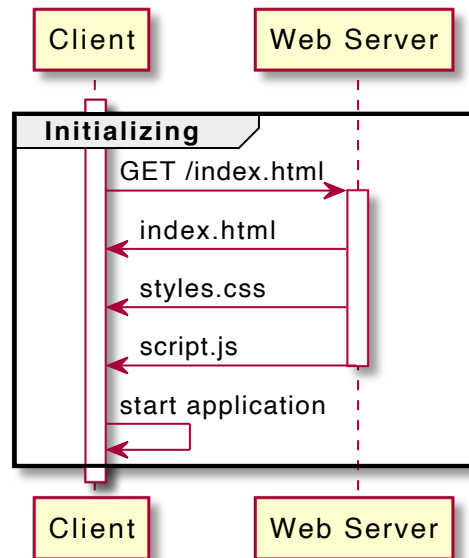
Speaker notes

- Wenn es mehrere Instanzen eines Backendservices geben soll, muss ein Loadbalancing die Anfragen der Nutzer auf die einzelnen Instanzen verteilen
- Dies wird einfacher durch eine Clientseitige Session, da der Client nicht immer zum selben Service geroutet werden muss

NACHTEILE EINER SPA

- initiale Response ist groß
- Client ist nicht Vertrauenswürdig
- duplizierter Code
- höherer Entwicklungsaufwand

INITIALE RESPONSE IST GROSS



Speaker notes

- Wir erinnern uns an das SPA Laufzeitdiagramm
- zu Anfang müssen erstmal alle Daten geladen werden

CLIENT IST NICHT VERTRAUENSWÜRDIG

- JavaScript Code auf dem Client kann manipuliert werden
- erneute Validierung im BE notwendig
- Validierungen sind meist duplizierter Code
- dadurch entsteht mehr Entwicklungsaufwand
- hierfür gibt es Abhilfe:

Speaker notes

- ein versierter Nutzer kann den JavaScript Code in seinem Browser verändern.
 - wir sprechen hier noch nicht mal von XSS
- Daten die im BE gespeichert werden, müssen daher noch mal validiert werden
- Validierungen, aber auch andere Geschäftslogik sind häufig dupliziert.
 - Das kann gewünscht sein. Vielleicht möchte man Frontend und Backend voneinander entkoppeln
 - Andererseits kann man Code auch übers BE und Frontend sharen.
 - Multiplattform Libraries wie von Kotlin können hier helfen

WIE BAUT MAN EINE SPA?

EINFACH MAL LOSLEGEN?

EINFACH MAL LOSLEGEN?

- Erster Gedanke: Einfach mal loslegen.
- Wie soll die UI aussehen?
- Welche HTML Elemente brauche ich?
- Was brauche ich fürs Styling?
- Welche Logik soll das Frontend unterstützen?

Speaker notes

- Vielleicht denken sie hier noch an die Trennung von HTML, CSS und JavaScript.
- Damit wird das ganze dann ein wenig schöner.
- Trotzdem wird es wahrscheinlich damit Enden, dass man einen Monolith erhält.

MONOLITH



Speaker notes

- Sieht für sie jetzt evtl. schön aus.
- Aber versuche sie mal diese Anwendung wiederzuverwenden oder sogar für neue Features zu erweitern.
- Das ist meistens das Problem bei monolithischen Anwendungen.

MONOLITH

- Monolithen sind typischerweise:
 - schwer wiederverwendbar
 - schwer erweiterbar
- Monolithen haben in sich meist:
 - keine klaren Schnittstellen
 - viele Abhängigkeiten

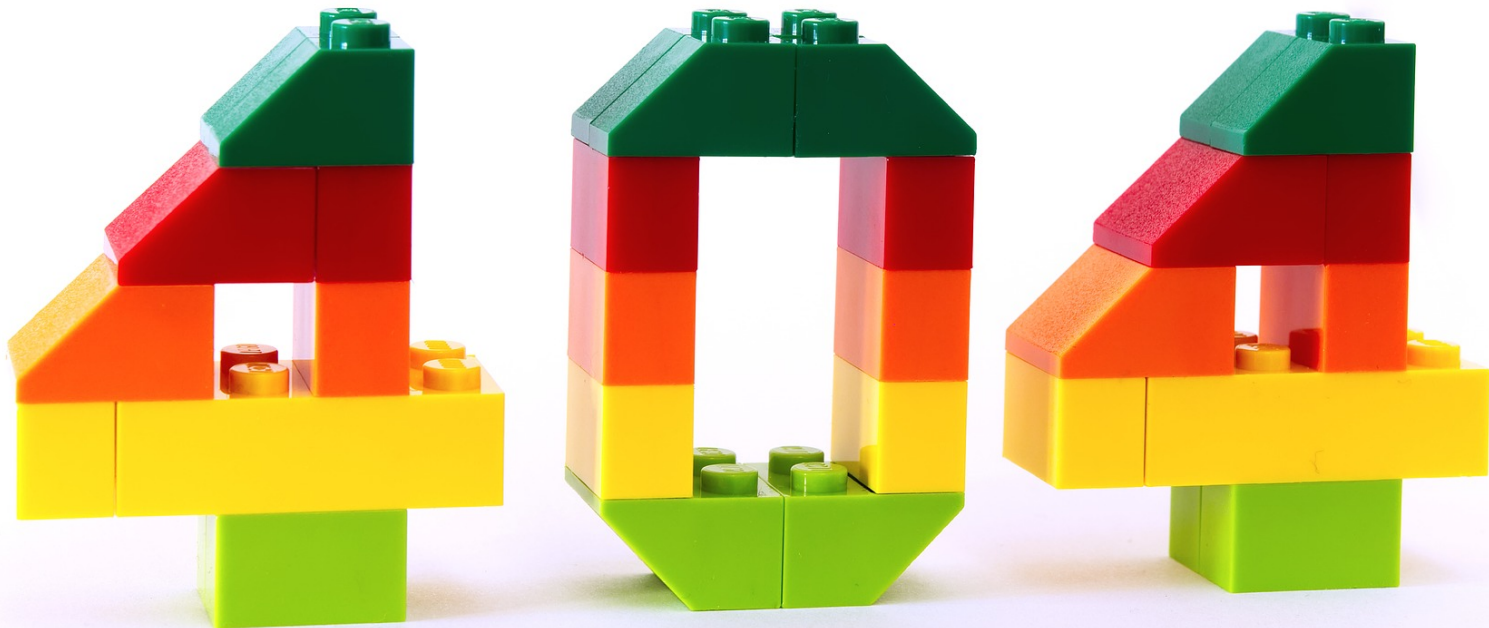
NICHT FALSCH VERSTEHEN

- ein Monolith ist zum starten erstmal sinnvoll
- ein Monolith kann durchaus seine Berechtigung haben
- mit wachsender Codebasis wird es unübersichtlich
- mit mehreren Teams an einem Monolith treten Konflikte auf

Speaker notes

- Monolithen sind nicht grundsätzlich schlecht
- Ein gut designer Monolith kann zu einem Modulith werden und sehr gut funktionieren

COMPONENT ARCHITECTURE



Speaker notes

- Component Architecture könnte man sich wie Lego vorstellen

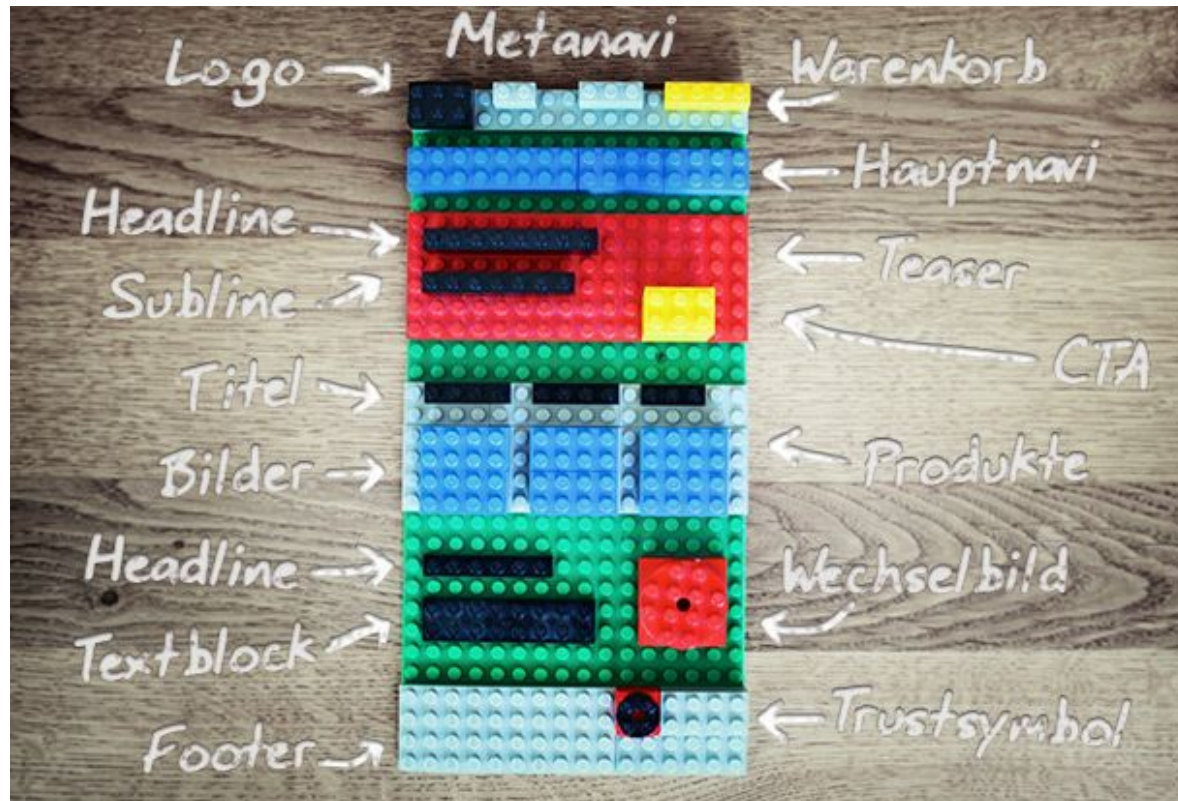
COMPONENT ARCHITECTURE

- divide et impera
 - teilen der Webseite in einzelnen Components
 - Verteilung und Strukturierung der Komplexität
- Components
 - enthalten zusammengehörige Funktionalität
 - quasi wie Klassen in OOP
 - haben feste Schnittstellen
 - möglichst lose Kopplung und hohe Kohäsion

Speaker notes

- divide et impera: steht natürlich nur auf der Folien, weil lateinische Wörter klug aussehen
- Die Idee ist aber grundlegend seine Webseite in einzelne Components aufzuteilen.
- Damit teilt man die Komplexität seiner Seite in kleinere Teile (Components).
- Quasi wie man es aus dem klassischen Softwareengineering kennt. Dort wird auch funktionalität die zusammengehört in Klassen zusammengefasst.
- Über Schnittstellen (vergleich zu Lego die Noppen), können Components dann wieder zusammengesteckt werden.

COMPONENT ARCHITECTURE



COMPONENT ARCHITECTURE

- SRP: Single Responsible Principle
- *"A class should have only one reason to change."*
- *"A module should be responsible to one, and only one, actor."*
- dies ist auch auf Components anwendbar
- Components sollten
 - nur einen Grund haben sich zu ändern
 - nur einem Akteur gegenüber verantwortlich sein

Speaker notes

- Zitate von Robert C. Marting SOLID und Clean Architecture
- Man könnte sich denken, dass dies nur bei kleineren Components möglich ist
- Doch auch eine Page hat eine Verantwortlichkeit und damit nur einen Grund sich zu ändern
- Bzw. sie ist gegenüber einem Akteur verantwortlich

COMPONENT ARCHITECTURE

- Was könnte man sich alles als Component vorstellen?
 - Buttons, Text Fields, Labels, etc.
 - Search Bar, Form Groups, Cards, etc.
 - Header, Footer, Overlays, etc.
 - Pages

Speaker notes

- Unter einer Card kann man sich gebündelten Content vorstellen. Möglicherweise mit Bild und Edit Button oder so?
- Eine Component kann also ein sehr kleiner Teil der Anwendung sein, wie z.B. ein einzelner Button
- Eine Component kann aber auch ein Abschnitt sein oder sogar eine ganze Seite, die sich mit einem bestimmten Thema beschäftigt.

COMPONENTS

```
1 <button value="Submit" onclick="alert('Button clicked!')"/>
```

- Components haben wie Classes feste Schnittstellen
- damit können sie modular eingesetzt werden
- normalerweise gibt es Input und Output Parameter

Speaker notes

- normalerweise gibt man etwas in eine Component hinein und bekommt etwas aus der Component zurück
- wir setzen mit der Component Architecture auf klassischen HTML Elementen auf und bauen daraus größere Components

COMPONENTS

Beispiel (Angular):

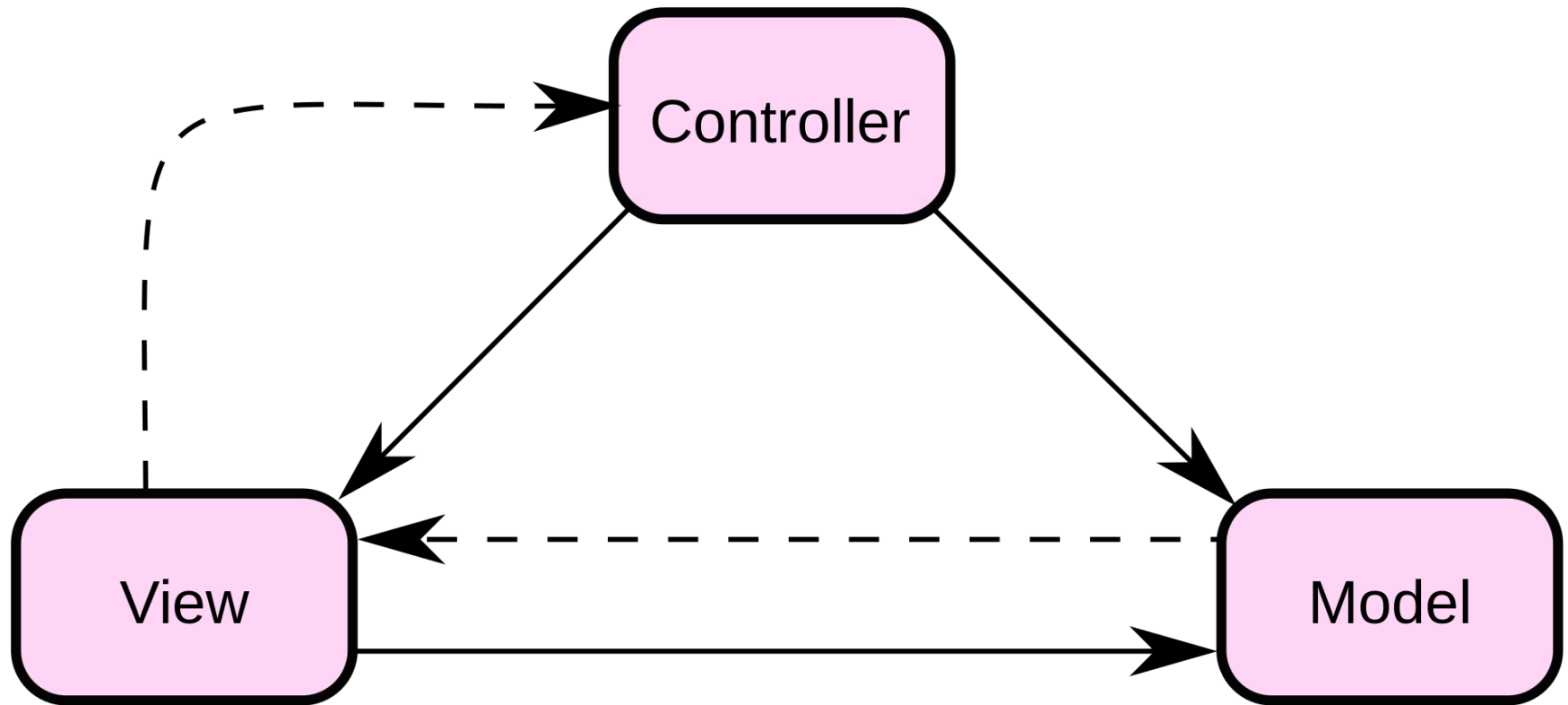
```
1 export class TextInputComponent {
2
3     @Input()
4     placeholder: string;
5     @Output()
6     text: EventEmitter<string>;
7
8     someLogic() {
9         // some logic
10    }
```

Speaker notes

- Jetzt fragen sie sich vielleicht, warum sollte ich eine TextInputComponent selbst bauen? Die gibts doch schon in HTML?
- Components abstrahieren nicht nur Struktur (HTML) sondern liefern gleich auch das Styling mit.
- der Aufrufer einer Component soll lediglich Input und Output mitgeben und sich um Struktur und Styling keine Gedanken machen müssen.

AUFBAU EINER COMPONENT ARCHITECTURE

parallelen zum MVC Pattern



Speaker notes

- um dies besser zu verstehen schauen wir uns dies anhand eines Angular Beispiels an

AUFBAU EINER COMPONENT: BEISPIEL ANGULAR

Speaker notes

- grundsätzlich besteht eine Component aus getrenntem Typescript, HTML und CSS
- die Trennung macht eine Component in sich übersichtlich
- es empfiehlt sich möglichst wenig Logik im HTML zu hinterlassen, dafür ist das Typescript File
- die html Datei könnte man daher als View bezeichnen
- das Model sind einfach die Klassen, die im View angezeigt werden
- einen klassischen Controller oder ein View Model gibt es nicht wirklich
- es gibt aber klare parallelen zwischen einer Component Architecture und einem MVC Pattern
- Ich gehe später noch mal genauer auf die einzelne Syntax etc. von Angular ein, damit ihr auch ein Praxisbeispiel dazu bauen könnt.
- <https://stackoverflow.com/questions/36950582/angular-design-pattern-mvc-mvvm-or-mv>

COMPONENT ARCHITECTURE

- Vorteile:
 - Konsistenz im Styling

Speaker notes

- Konsistenz
 - Komponenten wie Buttons gehören zu Atomen und sollten wiederverwendet werden.
 - Dies spart Zeit, außerdem sehen die Button überall gleich aus. Sorgt für Konsistenz im Styling
- Schnellere Entwicklung
 - Ich muss den Button nicht noch mal für eine andere Seite Stylen oder mit den Code dazu kopieren.
 - Ich kann auf bereits basierende Strukturen aufbauen.
- tiefe Verschachtelungen
 - Große Seiten und Anwendungen kämpfen häufig mit einer sehr hohen Verschachtelungstiefe
 - Durch Komponenten die kein Styling hinzufügen, sondern nur Logik bereitstellen und teilen, wird die Wrapper Hölle noch schlimmer.
 - Dies ist nicht sehr übersichtlich.
- Logik in Components
 - View Components sollten relativ frei von Logik sein.
 - Logik sollte in Services oder ähnliches ausgelagert werden.

COMPONENT ARCHITECTURE FRAMEWORKS

- Angular
- React
- Vue

Speaker notes

- die meisten JavaScript SPA Frameworks setzen auf eine Component Architecture.
- die Frameworks unterscheiden sich meistens nur in Details, Syntax, Performance.
- hat man die Basis, also Component Architectures verstanden, so kann man sich leicht an neue Frameworks gewöhnen
- kann man eins, kann man alle...
- es gibt allerdings doch einige unterschiede, die wir uns jetzt anschauen wollen.
- Dazu könnt ihr euch das mal ansehen: <https://academind.com/tutorials/angular-vs-react-vs-vue-my-thoughts/>
6
- wir schauen uns später konkret Angular an.

ANGULAR

- mehr eine Plattform als ein Framework
- kann einiges "out of the box"
 - DOM Manipulation
 - State Management
 - Routing
 - Form Validation
 - HTTP Client

Speaker notes

- bringt fast alles mit was man braucht
- Vorteil: Versionen sind im Angular Ökosystem kompatibel
- kann jedoch trotzdem um Libraries erweitert werden

REACT

- sehr leichtgewichtig
- reduziert auf
 - DOM Manipulation
 - State Management
- nur die Basis für die Component Architecture
- erweiterbar über Libraries

Speaker notes

- sehr leichtgewichtig
- Libraries durch eine große Community
- Versionsproblematik mit externen Libraries

VUE

- liegt zwischen Angular und React
- bietet
 - DOM Manipulation
 - State Management
 - Routing

EXKURS: ATOMIC DESIGN

EXKURS: ATOMIC DESIGN

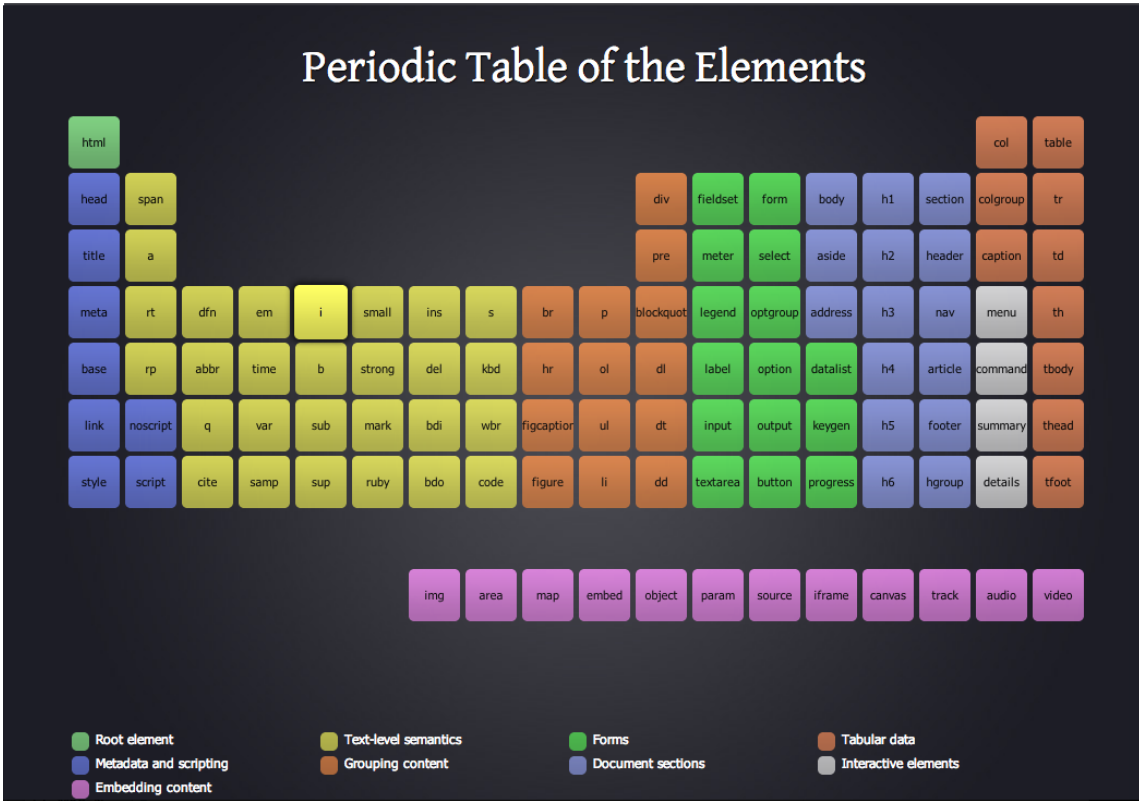
große Frontends mit vielen Components werden unübersichtlich



EXKURS: ATOMIC DESIGN

Strukturierung und Kategorisierung von Components

Ziel ist ein ordentlicher Baukasten an Components



<https://bradfrost.com/blog/post/atomic-web-design/>

EXKURS: ATOMIC DESIGN

- nach Atomic Design werden Components geordnet nach:
 - Atoms - Buttons, Text Fields, etc.
 - Molecules - Search Bar, Form Groups, etc.
 - Organisms - Header, Footer, Overlays, etc.
 - Templates - Schablone
 - Pages - konkrete Seite

Speaker notes

- <https://bradfrost.com/blog/post/atomic-web-design/>
- Atoms - die Bausteine unserer Anwendung - Buttons, etc.
- Molecules - kleine Zusammenschlüsse von Atoms - Suchfelder, Form Groups
- Organisms - fachliche Components. Zusammenschlüsse von Molecules mit denen der User interagieren kann
- Templates - Schablone die den Aufbau der Seite zeigt
- Pages - konkrete Seiten

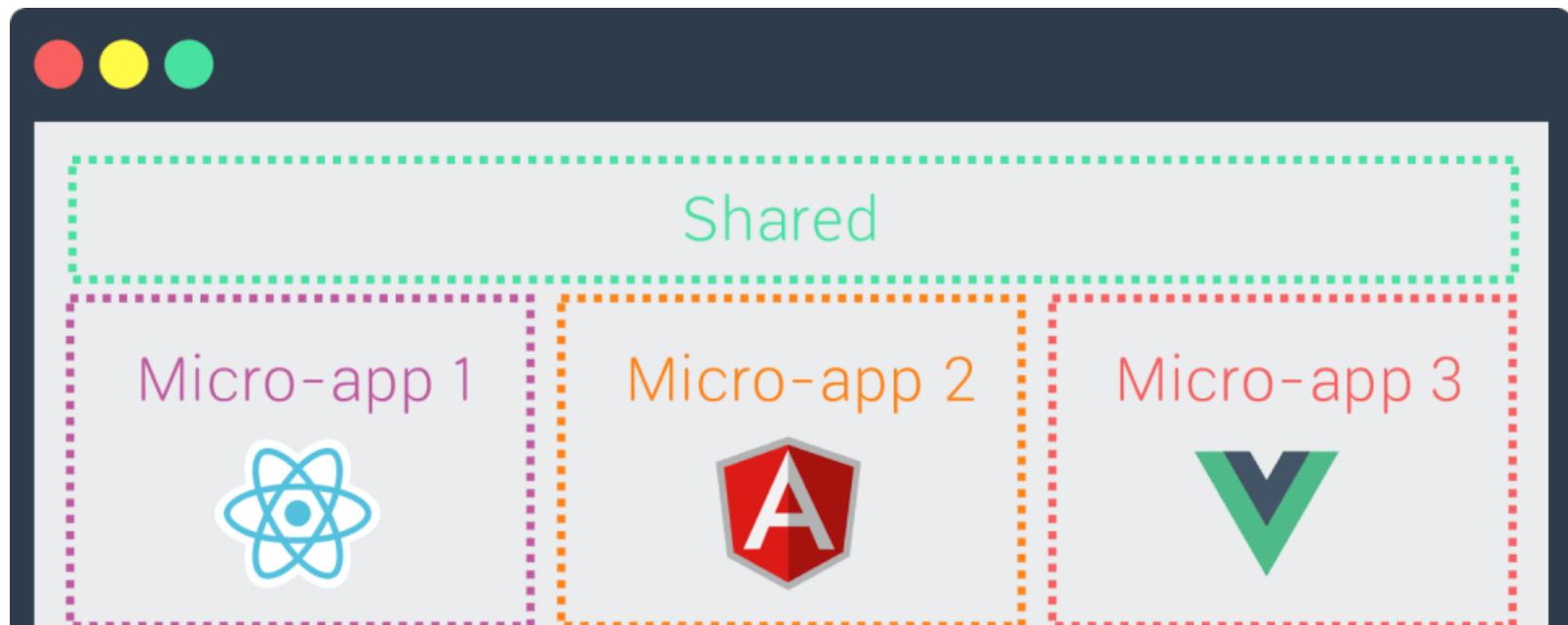
MICRO FRONTENDS

SPA & COMPONENT ARCHITECTURE

- bewahren uns nicht vor einem Monolith (Modulith)
- mehrere Teams an einem Monolith führt zu Konflikten
- schlechte Skalierbarkeit, wenn das Projekt wächst

MICRO FRONTENDS

- aufteilen des Monolith in mehrere Frontends
- Frontends können zu einem Frontend zusammengesteckt werden

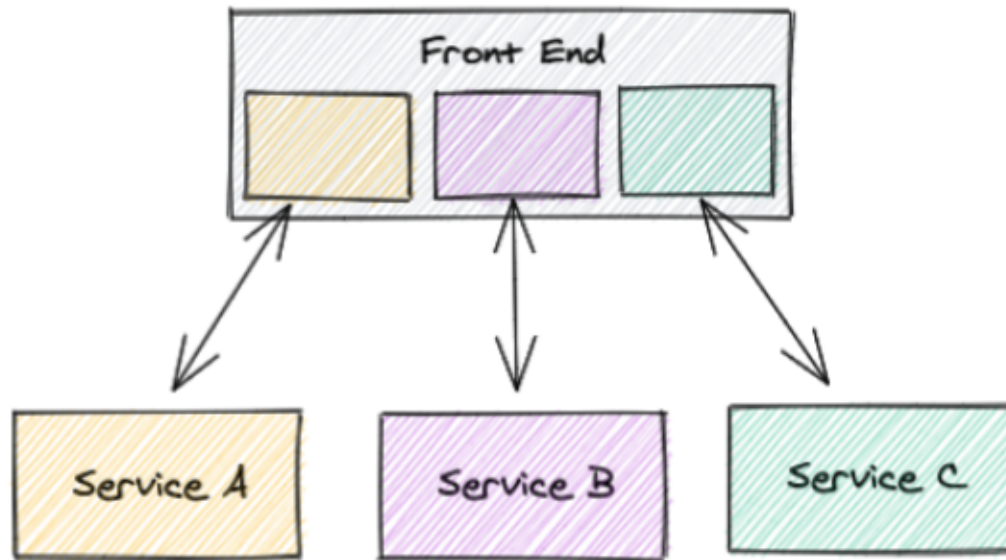


Speaker notes

- Micro Frontends können bei bedarf zu einem Frontend zusammengesteckt werden
- dies muss aber nicht sein, evtl. wird über eine Navigation von einem zum anderen Frontend navigiert
- Micro Frontends können von verschiedenen Teams mit verschiedenen Sprache und Frameworks gebaut werden
- evtl. auch ein eigener Deploy Zyklus

MICRO FRONTENDS

- reden meist auch mit eigenen Backends
- Micro Services

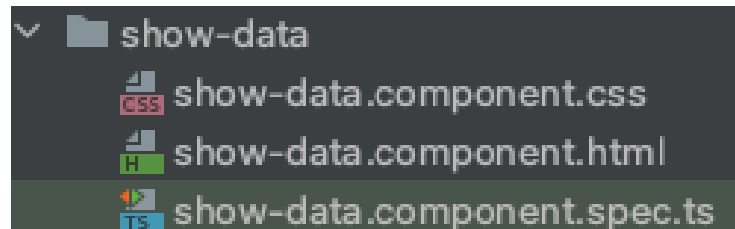


Speaker notes

- Micro Frontends sind aus Micro Services entstanden
- um auch die Frontends Skalierbarer zu machen
- dies ist nur bei wirklich komplexen Anwendungen zu empfehlen

ANGULAR

COMPONENT STRUCTURE



Speaker notes

- eine Angular Component besteht normalerweise aus 4 Dateien
- css -> enthält Informationen fürs Styling
 - Styling wird nur auf die eigene Component angewendet
 - Übergreifende Styles sollten an anderer Stelle abgelegt werden
 - Alternativ kann man die "ViewEncapsulation" auch ausschalten
- html -> enthält die Struktur des views
 - im HTML kann auf Variablen und Methoden der ts Datei zugegriffen werden
- spec.ts -> Tests für die Methode, wir gehen hier evtl. noch mal später ein
- ts -> enthält Logik und Daten

COMPONENT.TS

```
1 @Component({
2     selector: 'app-show-data',
3     templateUrl: './show-data.component.html',
4     styleUrls: ['./show-data.component.css']
5 })
6 export class ShowDataComponent {
7     ...
8 }
```

Speaker notes

- @Component Annotation
- selector -> damit wird die Component an anderer Stelle eingebunden
- templateUrl -> verweis auf das zugehörige HTML file
- styleUrls -> enthält ein Array von css files, die eingebunden werden
- die Component ist eine Typescript class

COMPONENT.TS

```
1 @Component({
2     selector: 'app-show-data',
3     templateUrl: './show-data.component.html',
4     styleUrls: ['./show-data.component.css']
5 })
6 export class ShowDataComponent {
7     ...
8 }
```

Speaker notes

- @Component Annotation
- selector -> damit wird die Component an anderer Stelle eingebunden
- templateUrl -> verweis auf das zugehörige HTML file
- styleUrls -> enthält ein Array von css files, die eingebunden werden
- die Component ist eine Typescript class

COMPONENT.TS

```
1 @Component({
2     selector: 'app-show-data',
3     templateUrl: './show-data.component.html',
4     styleUrls: ['./show-data.component.css']
5 })
6 export class ShowDataComponent {
7     ...
8 }
```

Speaker notes

- @Component Annotation
- selector -> damit wird die Component an anderer Stelle eingebunden
- templateUrl -> verweis auf das zugehörige HTML file
- styleUrls -> enthält ein Array von css files, die eingebunden werden
- die Component ist eine Typescript class

COMPONENT.TS

```
1 @Component({
2     selector: 'app-show-data',
3     templateUrl: './show-data.component.html',
4     styleUrls: ['./show-data.component.css']
5 })
6 export class ShowDataComponent {
7     ...
8 }
```

Speaker notes

- @Component Annotation
- selector -> damit wird die Component an anderer Stelle eingebunden
- templateUrl -> verweis auf das zugehörige HTML file
- styleUrls -> enthält ein Array von css files, die eingebunden werden
- die Component ist eine Typescript class

COMPONENT.TS

```
1 @Component({
2     selector: 'app-show-data',
3     templateUrl: './show-data.component.html',
4     styleUrls: ['./show-data.component.css']
5 })
6 export class ShowDataComponent {
7     ...
8 }
```

Speaker notes

- @Component Annotation
- selector -> damit wird die Component an anderer Stelle eingebunden
- templateUrl -> verweis auf das zugehörige HTML file
- styleUrls -> enthält ein Array von css files, die eingebunden werden
- die Component ist eine Typescript class

INPUT/OUTPUT

```
1 export class ShowDataComponent {  
2  
3     @Input()  
4     someData: SomeData;  
5     @Output()  
6     output: EventEmitter = new EventEmitter<Output>();  
7 }
```

Speaker notes

- Daten müssen in Components hinein und heraus gegeben werden
- Dafür gibt es Input und Output Parameter

INPUT

- einfache Datentypen
- werden automatisch aktualisiert

```
1 export class ShowDataComponent {  
2  
3     @Input()  
4     someData: SomeData;  
5     @Output()  
6     output: EventEmitter = new EventEmitter<Output>();  
7 }
```

Speaker notes

- Sie werden von der Parent Component in die Child Component gegeben
- Wenn sich die Daten der Parent Component ändern, werden die Daten in den Child Components ebenfalls geupdated
- Darum kümmert sich Angular als Framework

OUTPUT

- EventEmitter für das Datum
- output wird durch emit() ausgelöst

```
1 export class ShowDataComponent {  
2  
3     @Input()  
4     someData: SomeData;  
5     @Output()  
6     output: EventEmitter = new EventEmitter<Output>();  
7 }
```

Speaker notes

- EventEmitter stelle die emit() function bereit
- die Parent Component kann dann eine Callback Function angeben, die auf emit getriggert werden soll

INPUT/OUTPUT - PARENT

```
1 <app-show-data  
2     [someData]="{ ... }"  
3     (output)="callOnOutput($event)">  
4 </app-show-data>
```

Speaker notes

- Eckige Klammern werden für Inputs
- Runde Klammern für Outputs verwendet
- Ein Parameter kann auch Input und Output gleichzeitig sein
- \$event enthält dann die Daten die in das emit() gegeben wurden

LIFECYCLE METHODS

- werden zu bestimmten Ereignissen aufgerufen

```
1 export class ShowDataComponent implements OnInit {  
2  
3     ngOnInit() {  
4         // do something on init  
5     }  
6 }
```

Speaker notes

- Warum brauchen wir ngOnInit, es gibt doch ein
- ngOnInit findet im lifecycle später statt, wenn das HTML bereits initialisiert wurde
- weitere Lifecycle Methods findet ihr im Web <https://angular.io/guide/lifecycle-hooks>

Speaker notes

- die component.html bildet die Struktur der Component
- sie enthält neben HTML auch Zugriffe auf Methoden und Daten der component.ts
- Datenzugriffe
 - in geschweiften Klammern kann aus dem HTML auf Daten der Component zugegriffen werden
- Structural Directives
 - Wie auch bei anderen Frameworks (JSF) gibt es bei Angular so genannte structural directives
 - ngIf prüft eine Condition und zeigt den HTML Block an oder eben nicht
 - hier wird geprüft, ob someData gesetzt ist
 - in Direktiven kann ohne geschweifte Klammern auf Daten der Components zugegriffen werden
 - es gibt natürlich noch weitere structural directives wie ngFor, etc.
 - können auch selbst geschrieben/erweitert werden
- Click
 - (click) ist das Äquivalent zu onclick in Angular
 - An den runden Klammern erkennt man, dass es ein output Wert ist
- ngModel
 - Input und Output in einem
 - mit ngModel können wir Daten direkt mit einem Input Feld verknüpfen
 - es gibt Alternativen, wie z.B. FormControl -> darauf gehen wir nicht weiter ein

Speaker notes

- die component.html bildet die Struktur der Component
- sie enthält neben HTML auch Zugriffe auf Methoden und Daten der component.ts
- Datenzugriffe
 - in geschweiften Klammern kann aus dem HTML auf Daten der Component zugegriffen werden
- Structural Directives
 - Wie auch bei anderen Frameworks (JSF) gibt es bei Angular so genannte structural directives
 - ngIf prüft eine Condition und zeigt den HTML Block an oder eben nicht
 - hier wird geprüft, ob someData gesetzt ist
 - in Direktiven kann ohne geschweifte Klammern auf Daten der Components zugegriffen werden
 - es gibt natürlich noch weitere structural directives wie ngFor, etc.
 - können auch selbst geschrieben/erweitert werden
- Click
 - (click) ist das Äquivalent zu onclick in Angular
 - An den runden Klammern erkennt man, dass es ein output Wert ist
- ngModel
 - Input und Output in einem
 - mit ngModel können wir Daten direkt mit einem Input Feld verknüpfen
 - es gibt Alternativen, wie z.B. FormControl -> darauf gehen wir nicht weiter ein

Speaker notes

- die component.html bildet die Struktur der Component
- sie enthält neben HTML auch Zugriffe auf Methoden und Daten der component.ts
- Datenzugriffe
 - in geschweiften Klammern kann aus dem HTML auf Daten der Component zugegriffen werden
- Structural Directives
 - Wie auch bei anderen Frameworks (JSF) gibt es bei Angular so genannte structural directives
 - ngIf prüft eine Condition und zeigt den HTML Block an oder eben nicht
 - hier wird geprüft, ob someData gesetzt ist
 - in Direktiven kann ohne geschweifte Klammern auf Daten der Components zugegriffen werden
 - es gibt natürlich noch weitere structural directives wie ngFor, etc.
 - können auch selbst geschrieben/erweitert werden
- Click
 - (click) ist das Äquivalent zu onclick in Angular
 - An den runden Klammern erkennt man, dass es ein output Wert ist
- ngModel
 - Input und Output in einem
 - mit ngModel können wir Daten direkt mit einem Input Feld verknüpfen
 - es gibt Alternativen, wie z.B. FormControl -> darauf gehen wir nicht weiter ein

Speaker notes

- die component.html bildet die Struktur der Component
- sie enthält neben HTML auch Zugriffe auf Methoden und Daten der component.ts
- Datenzugriffe
 - in geschweiften Klammern kann aus dem HTML auf Daten der Component zugegriffen werden
- Structural Directives
 - Wie auch bei anderen Frameworks (JSF) gibt es bei Angular so genannte structural directives
 - ngIf prüft eine Condition und zeigt den HTML Block an oder eben nicht
 - hier wird geprüft, ob someData gesetzt ist
 - in Direktiven kann ohne geschweifte Klammern auf Daten der Components zugegriffen werden
 - es gibt natürlich noch weitere structural directives wie ngFor, etc.
 - können auch selbst geschrieben/erweitert werden
- Click
 - (click) ist das Äquivalent zu onclick in Angular
 - An den runden Klammern erkennt man, dass es ein output Wert ist
- ngModel
 - Input und Output in einem
 - mit ngModel können wir Daten direkt mit einem Input Feld verknüpfen
 - es gibt Alternativen, wie z.B. FormControl -> darauf gehen wir nicht weiter ein

Speaker notes

- die component.html bildet die Struktur der Component
- sie enthält neben HTML auch Zugriffe auf Methoden und Daten der component.ts
- Datenzugriffe
 - in geschweiften Klammern kann aus dem HTML auf Daten der Component zugegriffen werden
- Structural Directives
 - Wie auch bei anderen Frameworks (JSF) gibt es bei Angular so genannte structural directives
 - ngIf prüft eine Condition und zeigt den HTML Block an oder eben nicht
 - hier wird geprüft, ob someData gesetzt ist
 - in Direktiven kann ohne geschweifte Klammern auf Daten der Components zugegriffen werden
 - es gibt natürlich noch weitere structural directives wie ngFor, etc.
 - können auch selbst geschrieben/erweitert werden
- Click
 - (click) ist das Äquivalent zu onclick in Angular
 - An den runden Klammern erkennt man, dass es ein output Wert ist
- ngModel
 - Input und Output in einem
 - mit ngModel können wir Daten direkt mit einem Input Feld verknüpfen
 - es gibt Alternativen, wie z.B. FormControl -> darauf gehen wir nicht weiter ein

SERVICES

- möglichst wenig Logik in den Components
- Business Logik gehört in Services
- Services
 - werden in Components injected
 - werden bei der Initialisierung automatisch erzeugt

```
1 @Injectable({  
2     providedIn: 'root'  
3 })  
4 export class SomeDataService {  
5     ...  
6 }
```

Speaker notes

- Wie greifen die Components auf Services zu?
- mit Injectable markiert man, dass ein Service per Dependency Injection in einer Component injected werden kann
- hier evtl. ein kleiner Exkurs in Dependency Injection?

SERVICE INJECTION

- Angular injected Services automatisch

```
1 export class ShowDataComponent {  
2  
3     constructor(  
4         private readonly someDataService: SomeDataService  
5     ) {  
6     }  
7  
8     updateData() {  
9         this.someDataService  
10            .updateSomeData(this.someData);  
11     }  
12 }
```

MODULE

größere Anwendungen können modularisiert werden

```
1 @NgModule({
2   declarations: [
3     AppComponent,
4     ShowDataComponent,
5     SomeOtherComponent,
6   ],
7   imports: [
8     AppRoutingModule,
9   ],
10  bootstrap: [AppComponent]
11 })
```

Speaker notes

- für kleinere Anwendungen braucht man nicht mehrere Module
- für größere Anwendungen bietet es sich aber an fachliche Schnitte zu machen
- declarations -> in Modulen werden die Components deklariert
- imports -> hier können andere Module importiert werden
- bootstrap -> initiale Component

ROUTING

- Routes werde im RoutingModule registriert
- RoutingModule wird im AppModule importiert

```
1  const routes: Routes = [  
2    {  
3      path: 'show-data',  
4      component: ShowDataComponent,  
5    },  
6  ];  
7  
8  @NgModule({  
9    imports: [RouterModule.forRoot(routes)],  
10   exports: [RouterModule]  
11 })  
12 export class AppRoutingModule {  
13 }
```

Speaker notes

- im RoutingModule werden die Routes (Pfade) und die zugehörigen Components verknüpft
- so kann der User auch über eine Url auf eine spezifische Seite gelangen

ROUTES

```
1 const routes: Routes = [  
2   {  
3     path: '',  
4     redirectTo: 'show-data',  
5   },  
6   {  
7     path: 'show-data',  
8     component: ShowDataComponent,  
9   },  
10  {  
11    path: 'other/:some-parameter-id',  
12    component: OtherComponent,  
13  },  
14 ];
```

Speaker notes

- ein Default Pfad '' sollte immer angegeben sein
- es können auch Pfad Parameter mitgegeben werden um auf eine spezielle Ressource zu verweisen

ROUTES

```
1 const routes: Routes = [  
2   {  
3     path: '',  
4     redirectTo: 'show-data',  
5   },  
6   {  
7     path: 'show-data',  
8     component: ShowDataComponent,  
9   },  
10  {  
11    path: 'other/:some-parameter-id',  
12    component: OtherComponent,  
13  },  
14 ];
```

Speaker notes

- ein Default Pfad '' sollte immer angegeben sein
- es können auch Pfad Parameter mitgegeben werden um auf eine spezielle Ressource zu verweisen

ROUTES

```
1 const routes: Routes = [  
2   {  
3     path: '',  
4     redirectTo: 'show-data',  
5   },  
6   {  
7     path: 'show-data',  
8     component: ShowDataComponent,  
9   },  
10  {  
11    path: 'other/:some-parameter-id',  
12    component: OtherComponent,  
13  },  
14 ];
```

Speaker notes

- ein Default Pfad '' sollte immer angegeben sein
- es können auch Pfad Parameter mitgegeben werden um auf eine spezielle Ressource zu verweisen

ROUTER OUTLET

- Platzhalter für das Routing

```
1 <!--app.component.html-->  
2 <router-outlet></router-outlet>
```

Speaker notes

- im router-outlet werden dann die Components die übers Routing erreicht werden eingefügt

INTERNES ROUTING

```
1 export class ShowDataComponent {  
2  
3     constructor(private readonly router: Router) {  
4     }  
5  
6     async navigateToOtherComponent() {  
7         await this.router.navigate(['other-component']);  
8     }  
9 }
```

Speaker notes

- der Pfad wird in einem Array in die Methode navigate hineingegeben
- der Pfad muss mit einer der Routen aus dem Routing Module übereinstimmen
- im Array können auch mehrere Strings übergeben werden. Diese werden einfach über ein / miteinander verknüpft

ROUTING MIT PARAMETERN

```
1 export class ShowDataComponent {
2
3     constructor(private readonly router: Router) {
4     }
5
6     async navigateToOtherComponent() {
7         await this.router
8             .navigate(
9                 [ 'other-component',
10                  'some-parameter' ]
11             );
12     }
13 }
```

Speaker notes

- Parameter werden einfach an das Array angehängt, schließlich werden sie einfach an den Pfad gehängt
- im Routing Module müssen Parameter speziell gekennzeichnet werden -> Erinnerung :some-parameter-id

AUSLESEN DES PARAMETER

```
1 export class OtherComponent {  
2  
3     constructor(activatedRoute: ActivatedRoute) {  
4         const someParameter =  
5             activatedRoute  
6                 .snapshot  
7                 .paramMap  
8                 .get( 'some-parameter-id' );  
9     }  
10 }
```

Speaker notes

- ActivatedRoute wird injected
- aus ihr kann der Parameter ausgelesen werden
- der Identifier ist aus den Route Definitionen aus dem Routing Module

HTTP CLIENT

- wird Injected
- kennt die HTTP Methods

```
1 export class SomeDataService {  
2  
3     constructor(private readonly httpClient: HttpClient) {  
4     }  
5  
6     getSomeData(): Promise<SomeData> {  
7         return this.httpClient  
8             .get<SomeData>('url')  
9             .toPromise();  
10    }  
11 }
```

Speaker notes

- über den HTTP Client können Backend Requests ausgeführt werden
- Üblicherweise zur Datenübertragung, kein HTML, Styles oder JS
- stellt Methoden für die HTTP Methods bereit, get, post, put, delete, etc.

HTTP CLIENT

- muss im Module importiert werden

```
@NgModule({  
  ...  
  imports: [  
    HttpClientModule,  
  ],  
  ...  
})  
export class AppModule {  
}
```

PRAXIS

PRAXIS: TODO ANWENDUNG

Speaker notes

- Wir haben uns jetzt grundlegende Konzepte für Frontends angesehen.
- Jetzt wird es Zeit für die Praxis.
- Dafür sollten wir uns noch mal die Syntax von Angular ansehen.

ANFORDERUNGEN

- List View mit allen Todo's
 - sortiert nach "done/undone"
 - "+" Button um neue Todo's hinzuzufügen
- Detail View
 - Detailansicht des Todo's
 - hier kann die "done" Checkbox bearbeitet werden
- Edit View
 - um neue Todo's anzulegen
 - und bestehende zu bearbeiten

TODO

```
1 export interface Todo {  
2     id: number;  
3     title: string;  
4     done: boolean;  
5 }
```

PRAXIS: LÖSUNGEN

- Branch: solution

BEIM NÄCHSTEN MAL:

- Statemanagement im Frontend
- Statemanagement mit Angular NgRx
- Vergleich zu anderen Frameworks
- Praxis: Einbau eines Statemanagements in unsere Todo Anwendung mit NgRx

WEITERE INFOS

- Component Architecture
 - <https://www.simform.com/blog/component-based-development/>
- Atomic Design
 - <https://bradfrost.com/blog/post/atomic-web-design/>
- Micro Frontends
 - <https://martinfowler.com/articles/micro-frontends.html>
 - <https://micro-frontends.org/>
 - <https://www.youtube.com/watch?v=BuRB3djraeM>
- Angular
 - <https://angular.io/>