

# **RICH CLIENT**

# WAS BEIM LETZTEN MAL GESCHAH

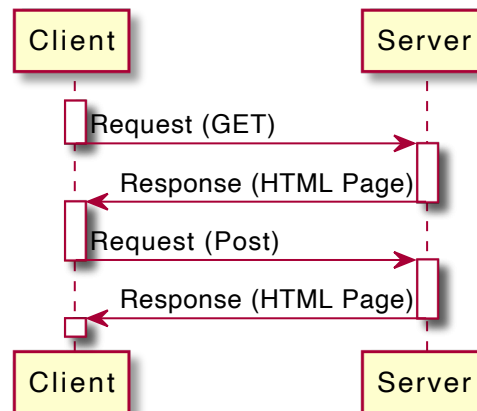
- Multi Page Application
- JSF
- Wiederholung MVC

# 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

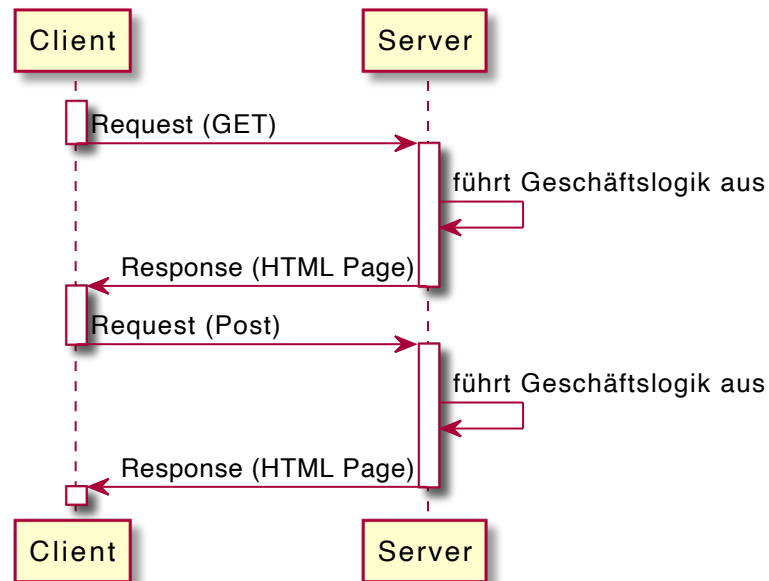
```
1 <html>
2   // head
3   <body>
4     <header></header>
5     <nav></nav>
6     <div>
7       // some data
8     </div>
9     <footer></footer>
10  </body>
11 </html>
```

# 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



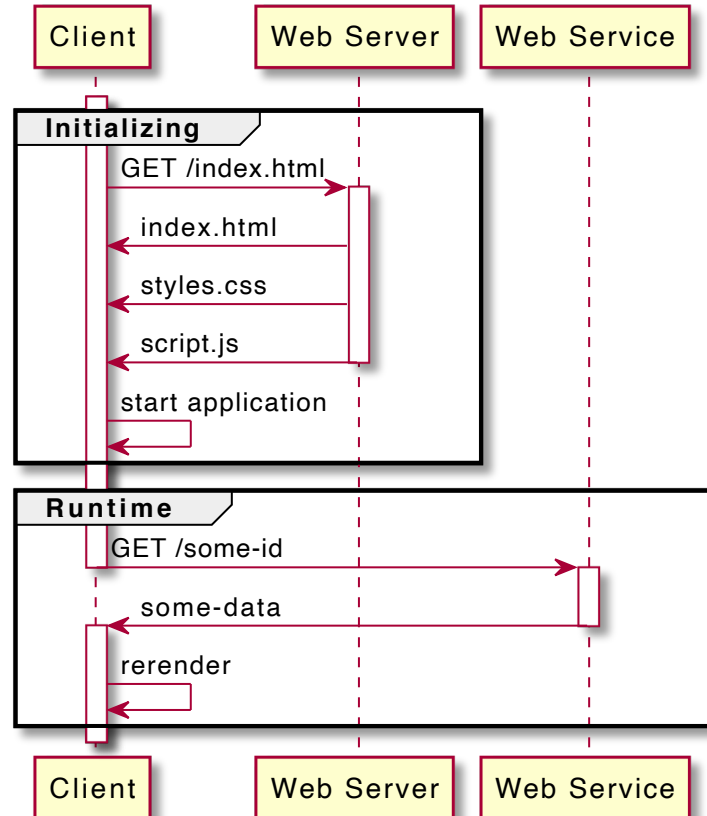
# **SINGLE PAGE APPLICATION**

*“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

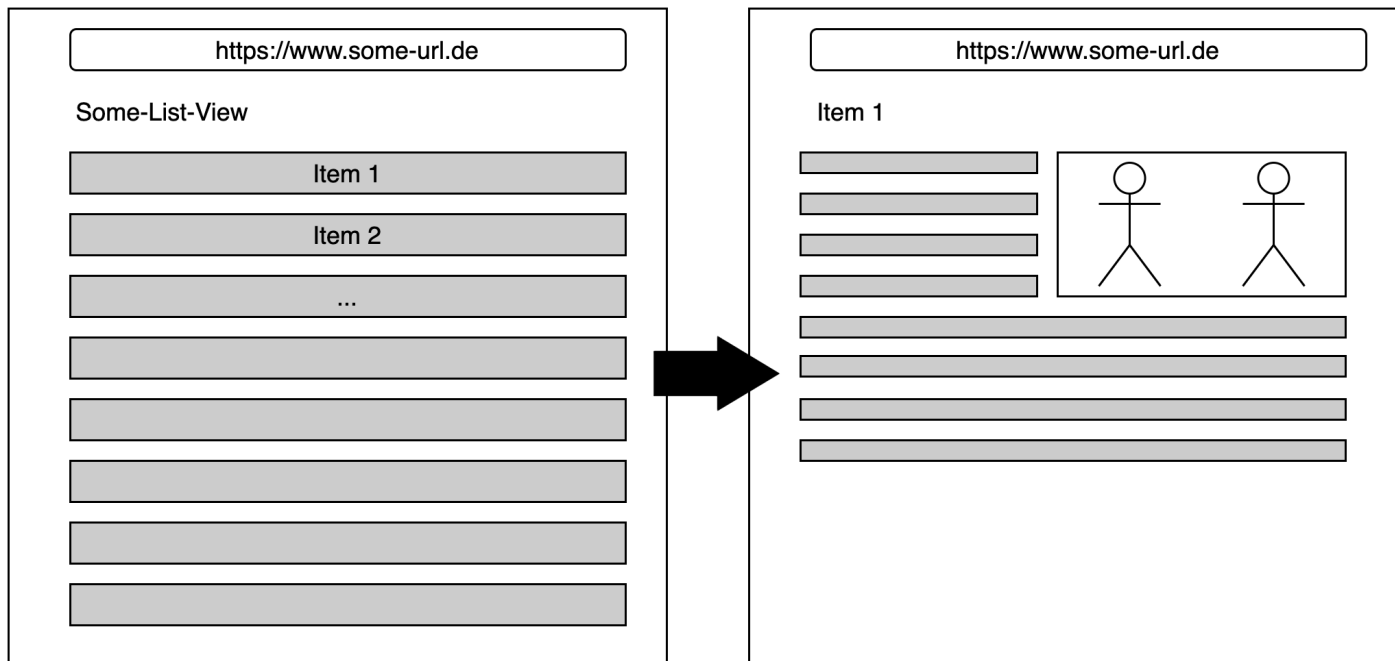
# SINGLE PAGE APPLICATION KONZEPT





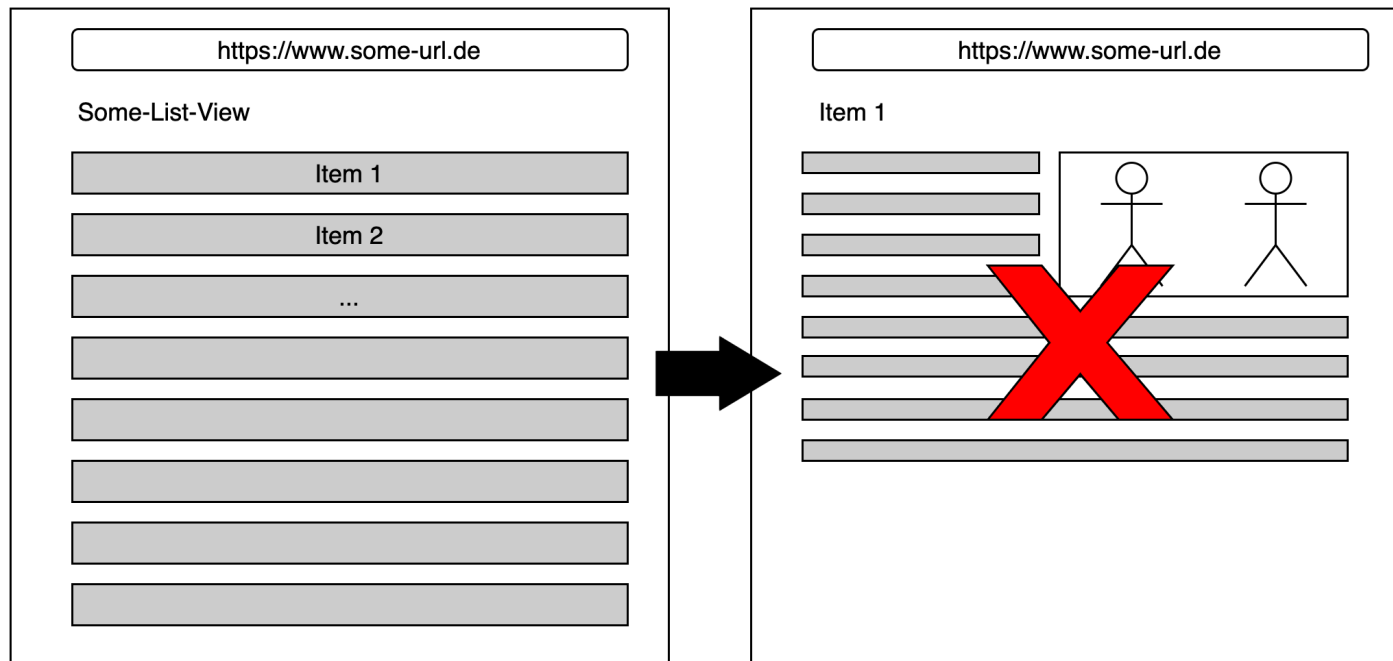
# ROUTING?

- ist eigentlich nicht notwendig
- Anwendung macht einfach ein rerender



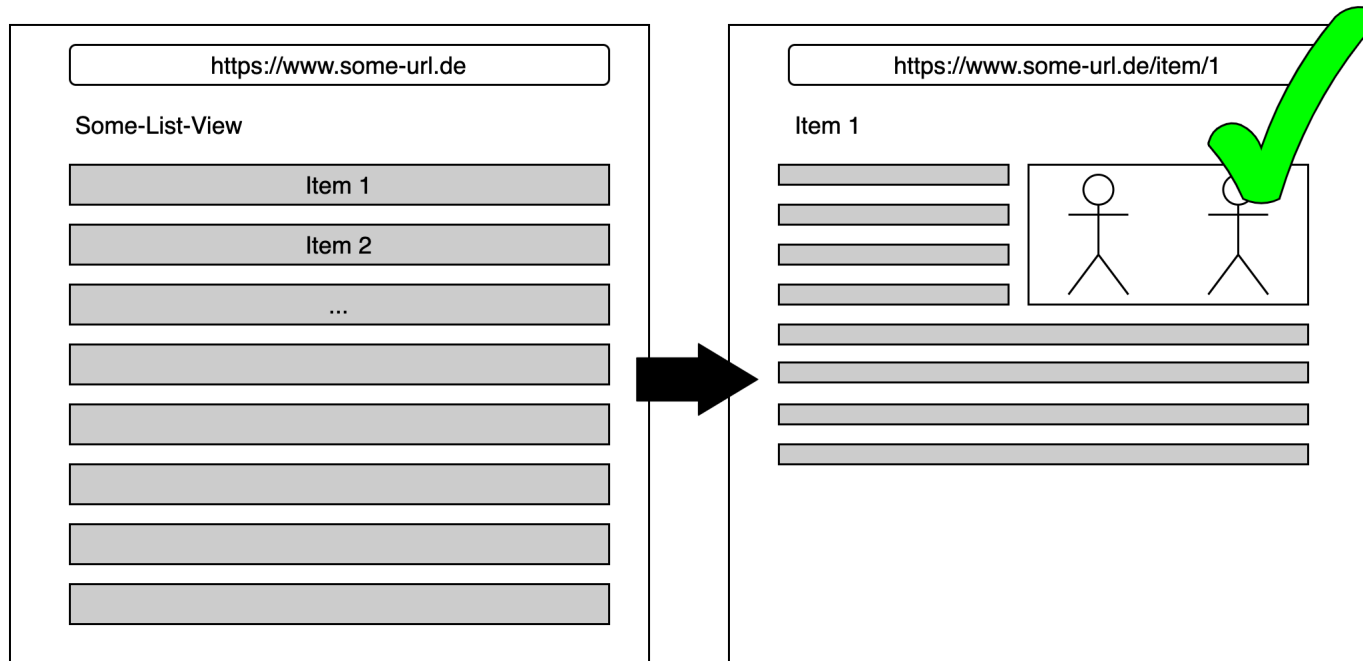
# ALSO KEIN ROUTING?

- 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

# 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>
```

## **BESSERE USER EXPERIENCE**

- kürzere Response Time
- weniger BE Request notwendig
- Seite ist während eines BE Requests benutzbar
- asynchrones Nachladen der Daten

## 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

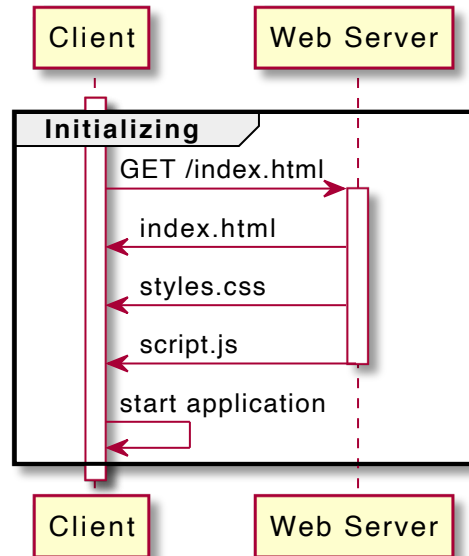
- Clientseitige Session
  - Server kann Stateless sein
  - Skalierbarkeit des Servers
  - Loadbalancing ist einfacher
- Hybride Anwendungen
  - SPA's sind ähnlich wie App's
  - React Native oder Flutter zeigen wie es geht



## NACHTEILE EINER SPA

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

# INITIALE RESPONSE IST GROSS



# 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:
  - Multiplattform Libraries

**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?

# MONOLITH



# MONOLITH

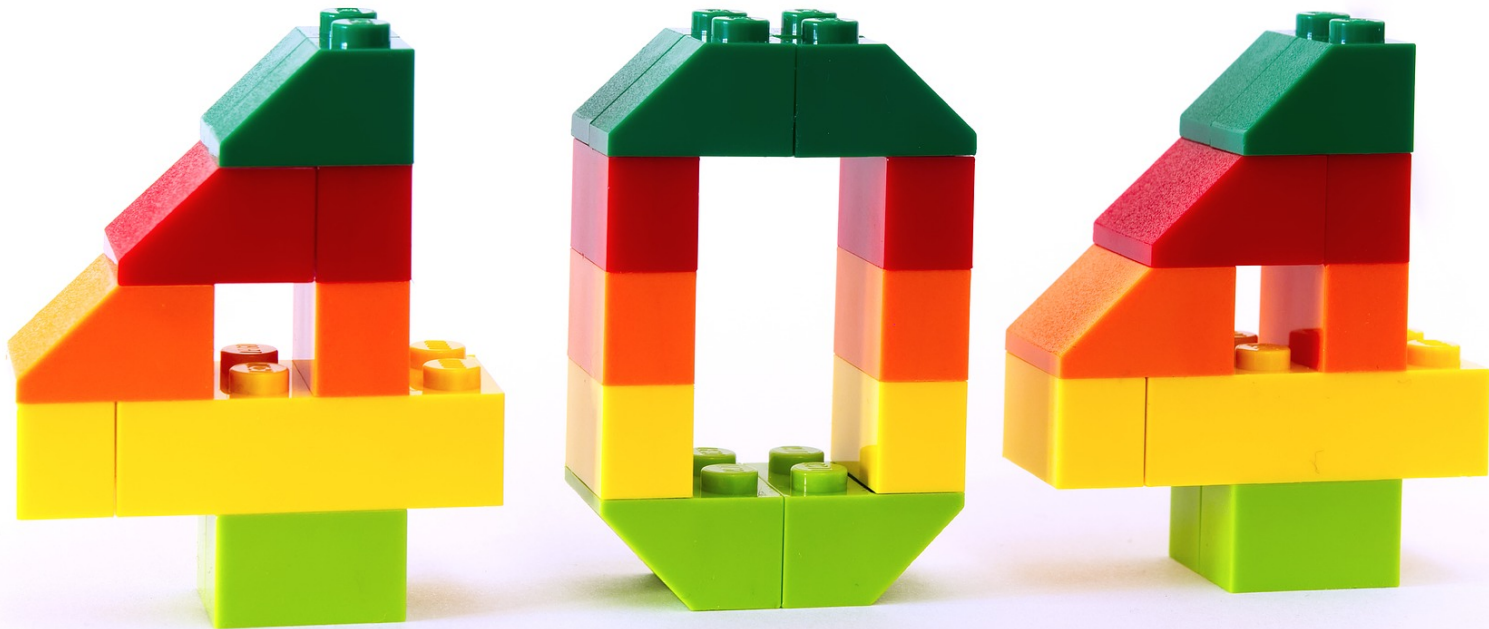
- Monolithen sind typischerweise:
  - schwer wiederverwendbar
  - schwer erweiterbar
- Monolithen haben in sich meist:
  - keine klaren Schnittstellen
  - viele Abhängigkeiten



## 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

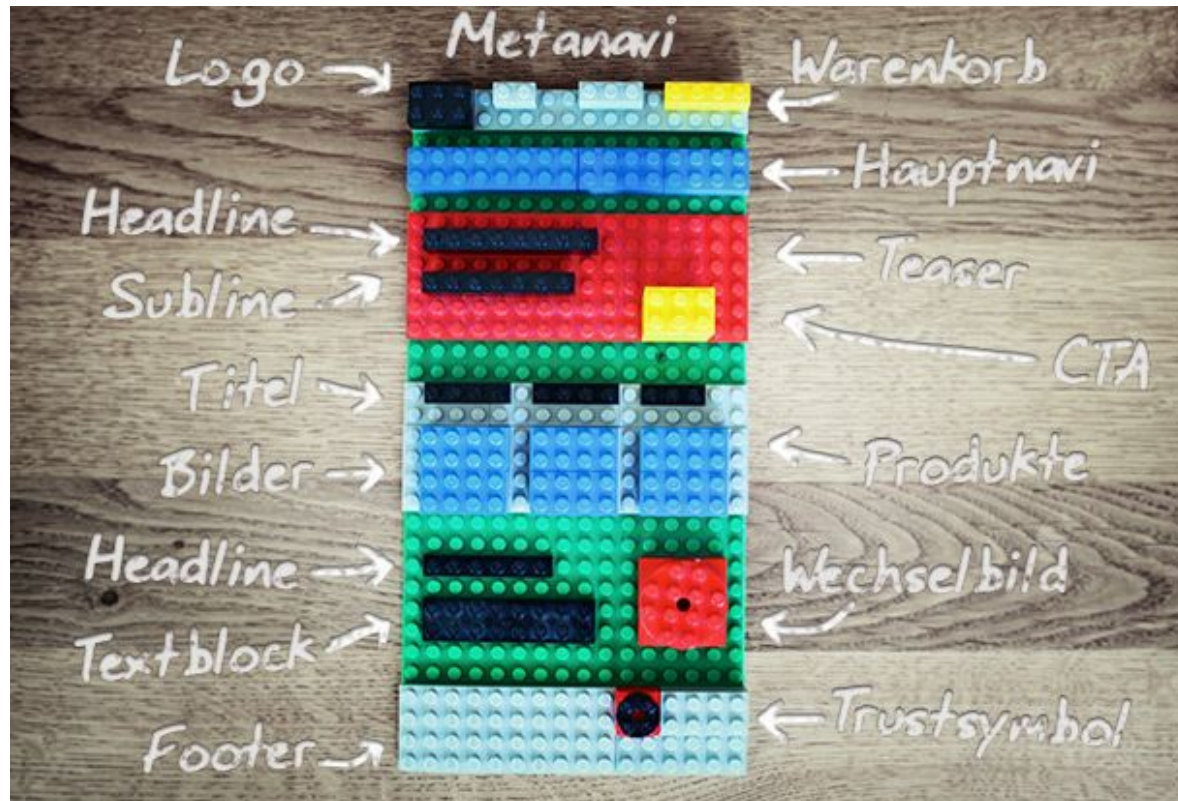
# COMPONENT ARCHITECTURE



# 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
    - analog wie Legosteine
  - abstrahieren Struktur und Styling

# 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

# 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

# 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

# COMPONENTS

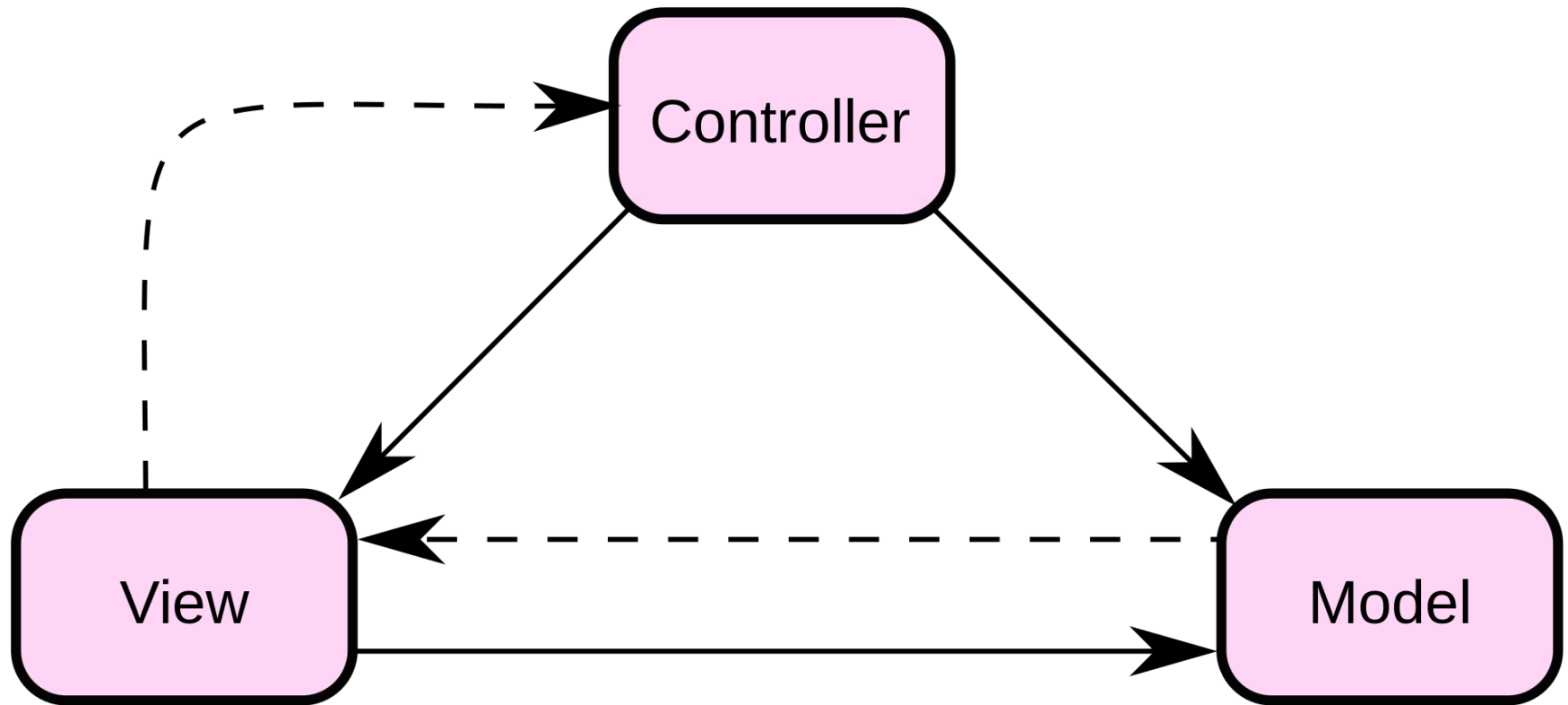
Beispiel (Angular):

```
1 export class TextInputComponent {
2
3     @Input()
4     placeholder: string;
5     @Output()
6     text: EventEmitter<string>;
7
8     someLogic() {
9         // some logic
10    }
11 }
```



# AUFBAU EINER COMPONENT ARCHITECTURE

parallelen zum MVC Pattern



# AUFBAU EINER COMPONENT: BEISPIEL ANGULAR

# COMPONENT ARCHITECTURE

- Vorteile:
  - Konsistenz im Styling
  - Wiederverwendbarkeit
  - schnellere Entwicklung
  - einfachere Instandhaltung
- Nachteile:
  - tiefe Verschachtelungen möglich
  - Logik in den Components
  - im Prinzip immer noch ein Monolith (Modulith)
  - mehrere Teams an einem Artefakt ist nicht optimal

# COMPONENT ARCHITECTURE FRAMEWORKS

- Angular
- React
- Vue
- und viele mehr...

# ANGULAR

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

# REACT

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

# VUE

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

# **EXKURS: ATOMIC DESIGN**



# EXKURS: ATOMIC DESIGN

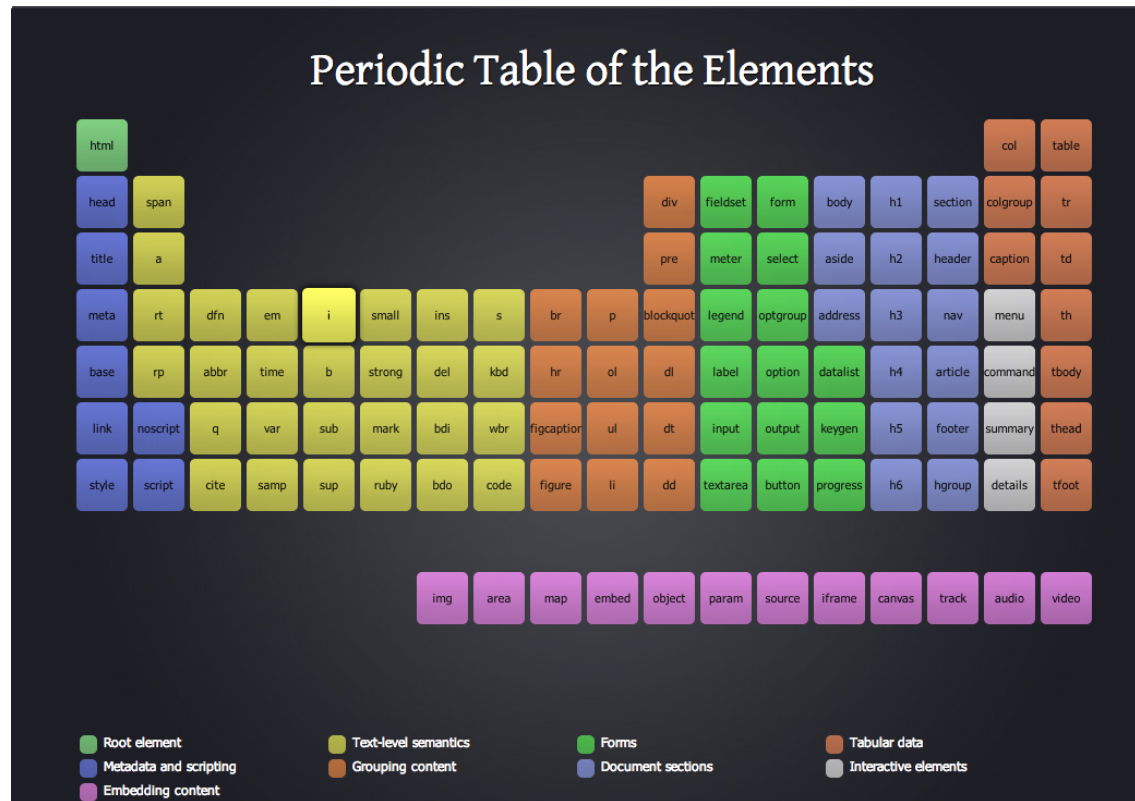
große Frontends mit vielen Components werden unübersichtlich



# 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

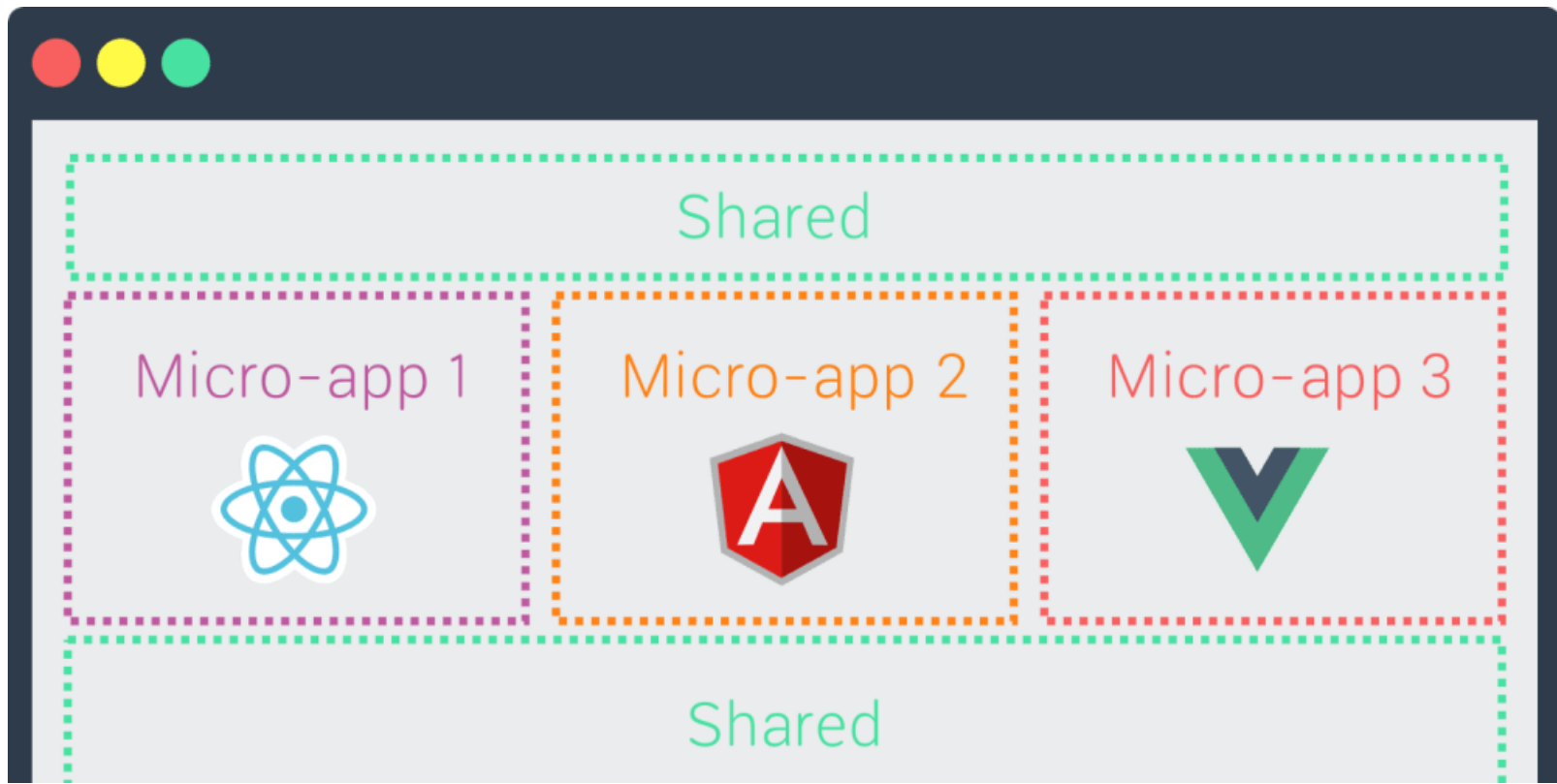
# MICRO FRONTENDS

# SPA & COMPONENT ARCHITECTURE

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

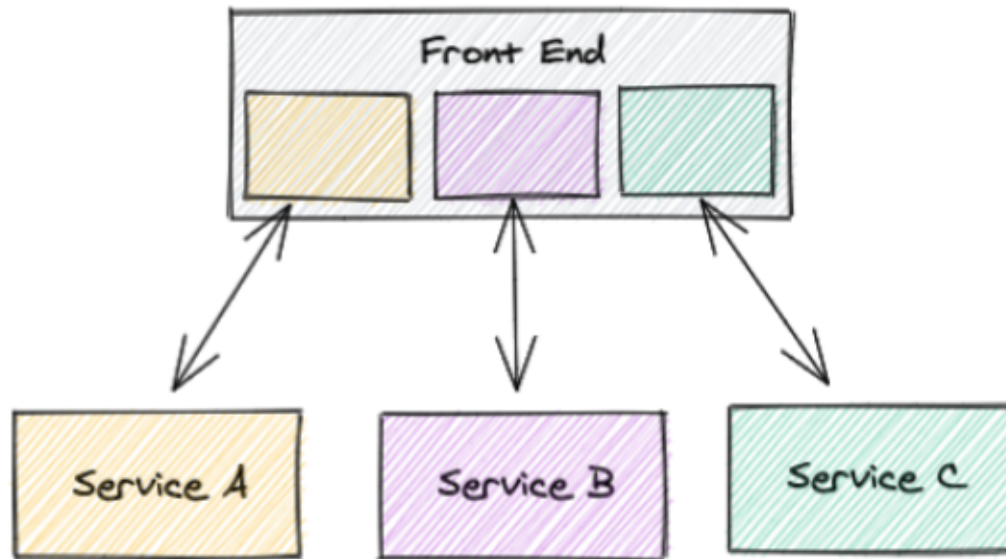
# MICRO FRONTENDS

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



# MICRO FRONTENDS

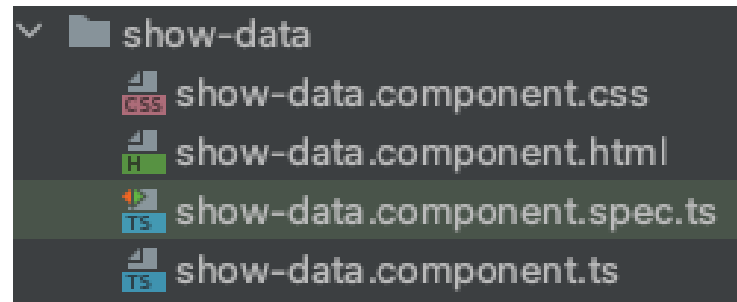
- reden meist auch mit eigenen Backends
- Micro Services



# ANGULAR



# COMPONENT STRUCTURE



# 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 }
```

# 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 }
```

# 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 }
```

# 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 }
```

# 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 }
```

# INPUT/OUTPUT

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

# 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 }
```



# 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 }
```

## INPUT/OUTPUT - PARENT

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

# LIFECYCLE METHODS

- werden zu bestimmten Ereignissen aufgerufen

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

# COMPONENT.HTML

```
1 <div *ngIf="someData">
2   <h4>{{someData.someTitle}}</h4>
3   <p>{{someData.someDescription}}</p>
4   <input type="text" [(ngModel)]="someData.text"/>
5   <input type="button" value="Submit"
6     (click)="updateData()" />
7 </div>
```

# COMPONENT.HTML

```
1 <div *ngIf="someData">
2   <h4>{{someData.someTitle}}</h4>
3   <p>{{someData.someDescription}}</p>
4   <input type="text" [(ngModel)]="someData.text"/>
5   <input type="button" value="Submit"
6     (click)="updateData()" />
7 </div>
```

# COMPONENT.HTML

```
1 <div *ngIf="someData">
2   <h4>{{someData.someTitle}}</h4>
3   <p>{{someData.someDescription}}</p>
4   <input type="text" [(ngModel)]="someData.text"/>
5   <input type="button" value="Submit"
6     (click)="updateData()" />
7 </div>
```

# COMPONENT.HTML

```
1 <div *ngIf="someData">
2   <h4>{{someData.someTitle}}</h4>
3   <p>{{someData.someDescription}}</p>
4   <input type="text" [(ngModel)]="someData.text"/>
5   <input type="button" value="Submit"
6     (click)="updateData()" />
7 </div>
```

# COMPONENT.HTML

```
1 <div *ngIf="someData">
2   <h4>{{someData.someTitle}}</h4>
3   <p>{{someData.someDescription}}</p>
4   <input type="text" [(ngModel)]="someData.text"/>
5   <input type="button" value="Submit"
6     (click)="updateData()" />
7 </div>
```



# 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 }
```

# 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 })
12 export class AppModule {
13 }
```

# 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 }
```

# 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 ];
```

# 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 ];
```

# 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 ];
```

# ROUTER OUTLET

- Platzhalter für das Routing

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



# 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 }
```

# 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 }
```

# 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 }
```

# 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 }
```

# HTTP CLIENT

- muss im Module importiert werden

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

# PRAXIS

# **PRAXIS: TODO ANWENDUNG**

# 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/>