



Angular

Teil 2

Agenda

Styling

Lifecycle-Hooks

Services

Routing

Rxjs für asynchrone Operationen

Styling

Angular Styles

app.component.ts

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})

export class AppComponent {
```

app.component.css

```
p {
  color: red;
  font-weight: 700;
}
```

app.component.ts

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  standalone: true,
  template: `HTML`,
  styles: [
    p {
      color: red;
      font-weight: 700;
    }
  ],
})

export class AppComponent {
```

```
}
```

Globale Styles

```
.  
├── public  
│   └── favicon.ico  
├── src  
│   ├── app  
│   │   ├── app.component.css  
│   │   ├── app.component.html  
│   │   ├── app.component.spec.ts  
│   │   ├── app.component.ts  
│   │   ├── app.config.ts  
│   │   └── app.routes.ts  
│   ├── index.html  
│   ├── main.ts  
│   └── styles.css  
├── README.md  
├── angular.json  
├── package-lock.json  
├── package.json  
├── tsconfig.app.json  
├── tsconfig.json  
└── tsconfig.spec.json
```

Tailwind CSS

- CSS Framework für schnelle Implementierung von UI
- Utility-First: Entwicklung mittels Hilfsklassen
- Umfangreiche Konfigurationsmöglichkeiten
- Keine vordefinierten Komponenten

Tailwind CSS - Setup

- Installation via npm
- Initialisierung via npx
- Verknüpfung mit Templates in generierter “tailwind.config.js”
- Integration in globale “styles.css”

```
npm install -D tailwindcss postcss autoprefixer
```

```
npx tailwindcss init
```

```
module.exports = {  
  content: [ "./src/**/*.{html,ts}", ],  
  theme: { extend: {}, },  
  plugins: [],  
}
```

```
@tailwind base;  
@tailwind components;  
@tailwind utilities;
```

Tailwind CSS - Verwendung

Styling mit Utility-Klassen:

app.component.html

```
<h1 id="heading" class="text-2xl font-bold text-white">  
  Hello World!  
</h1>
```

=

app.component.css

```
#heading {  
  font-size: 1.5rem;  
  line-height: 2rem;  
  font-weight: 700;  
  color: rgb(255,255,255);  
}
```

Alle Utility Klassen einfach zu finden in der Dokumentation
<https://tailwindcss.com/docs/installation>

Tailwind CSS - Beliebige Styles

- Beliebige Styles in Tailwind mit [] verwendbar

Beliebiger Wert

app.component.html

```
<h1 id="heading" class="text-[40px]">Hello World!</h1>
```

Beliebiges Property

app.component.html

```
<h1 id="heading" class="[font-size:20px]">Hello World!</h1>
```

Tailwind CSS - States

- Konditionelles Styling durch Präfix
- Präfixe können addiert werden

app.component.html

```
<h1 id="heading" class="bg-gray-300 hover:bg-green-600 dark:bg-white dark:hover:bg-green-300">  
  Hello World!  
</h1>
```

Beispiele

hover

focus

first-child

dark

first

last

Tailwind CSS - Responsive

Responsives Design ebenso durch Präfix

app.component.html

```
<h1 id="heading" class="text-md sm:text-xl hover:text-lg sm:hover:text-2xl">  
  Hello World!  
</h1>
```

Breakpoints:

- sm: 640px
- md: 768px
- lg: 1024px
- xl: 1280px
- 2xl: 1536px

Tailwind CSS - Theme

tailwind.config.js

```
/** @type {import('tailwindcss').Config} */
module.exports = {
  content: ["./src/**/*.{html,ts}"],
  theme: {
    screens: {
      sm: '480px',
      md: '768px',
      lg: '1028px',
    },
    extend: {
      fontFamily: {
        montserrat: ['Montserrat', 'sans-serif'],
      },
      colors: {
        primary: '#ea0b2a',
        secondary: '#920df1',
      },
    },
    plugins: [],
  };
}
```

- (fast) alles an Tailwind ist konfigurierbar
- Konfiguration durch Theme in “tailwind.config.js”
- Änderungen oder Erweiterungen möglich
- Achtung: bei direkter Anpassung (nicht extend) wird das komplette Objekt überschrieben

Tailwind CSS - Layer

styles.css

```
@tailwind base;  
@tailwind components;  
@tailwind utilities;  
  
@layer base {  
  body {  
    @apply font-sans;  
  }  
}
```

- Konfiguration auf den Tailwind-Ebenen
- `@layer` Syntax lässt CSS auf entsprechender Ebene anpassen
- `@apply` wendet gezielt Utility Klassen auf Eben an (auch Custom-Klassen aus Theme)

Aufgabe 2

10 Minuten

Lifecycle-Hooks

Lifecycle-Hooks

```
export class AppComponent implements  
OnChanges, OnInit, OnDestroy {  
  
  ngOnInit() {  
    ...  
  }  
  
  ngOnChanges() {  
    ...  
  }  
  
  ngOnDestroy() {  
    ...  
  }  
}
```

- Methoden, die zu bestimmten Zeitpunkten im Lebenszyklus einer Komponente ausgeführt werden:
 - ngOnInit: wenn Komponenten Inputs initialisiert wurden
 - ngOnChange: wenn sich Komponenten-Inputs verändern
 - ngOnDestroy: wenn die Komponente “destroyed” wird
 - ...

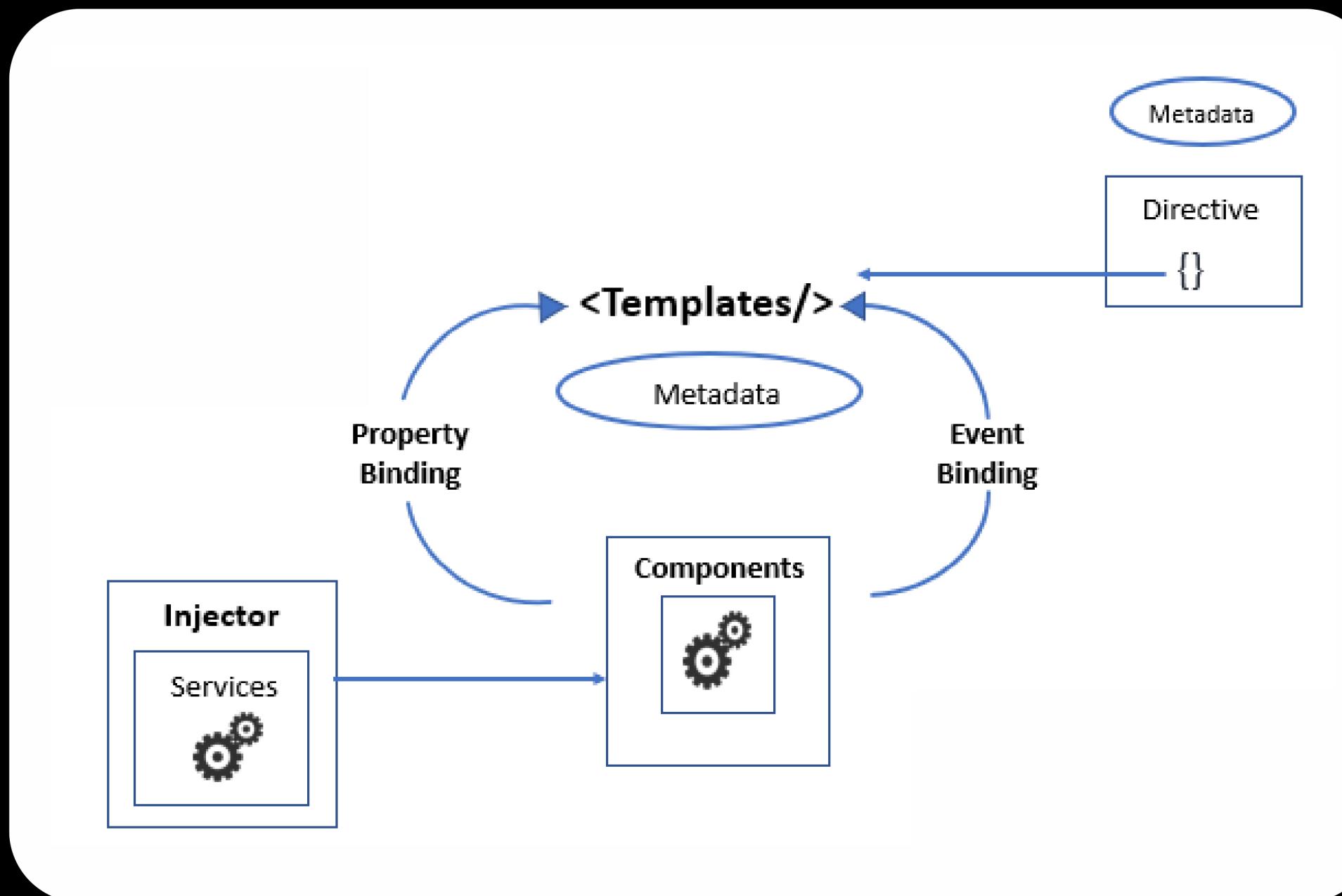
Services

Services

```
services
└── campus-event-data.service.spec.ts
└── campus-event-data.service.ts
```

- Kümmern sich um Backend-Requests und andere Geschäftslogik
- Werden von Angular verwaltet und als Singleton bereitgestellt
- Bestehen aus:
 - Logik - name.service.ts
 - Tests - name.service.spec.ts
- generierbar durch CLI

Services - Dependency Injection



- Injector stellt Singletons aller registrierten Services bereit
- Providers stellen dem Injector die Services bereit

Services - Erstellung

campus-event-data.service.ts

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class CampusEventDataService {
```

- Decorator für Konfiguration:
providedIn - Name des Injectors
- TypeScript Klasse für Logik:
API Requests
Geschäftslogik
Datenhaltung

Services - Verwendung

campus-event-list.component.ts

```
import { CampusEventDataService } from '../../../../../services/campus-event-data.service';
...

@Component({
  selector: 'app-campus-event-list',
  ...
})
export class CampusEventListComponent {
  ...
  constructor(private campusEventDataService: CampusEventDataService) {
    ...
  }
}
```

Aufgabe 3

15 Minuten

Routing

Routing

```
.  
├── public  
│   └── favicon.ico  
├── src  
│   ├── app  
│   │   ├── app.component.css  
│   │   ├── app.component.html  
│   │   ├── app.component.spec.ts  
│   │   ├── app.component.ts  
│   │   ├── app.config.ts  
│   │   └── app.routes.ts  
│   ├── index.html  
│   ├── main.ts  
│   └── styles.css  
└── README.md  
├── angular.json  
├── package-lock.json  
└── package.json  
├── tsconfig.app.json  
└── tsconfig.json  
└── tsconfig.spec.json
```

- Nicht file-based, sondern muss explizit definiert werden: *app.routes.ts*

Routing

app.routes.ts

```
export const routes: Routes = [
  { path: 'first-component', component: FirstComponent },
  { path: 'second-component', component: SecondComponent },
  { path: "other/:id", component: OtherComponent },
  { path: '**', component: PageNotFoundComponent },
];
```

app.component.html

```
<h2>Homepage</h2>

<nav>
  <ul>
    <li><a routerLink="first-component">First</a></li>
    <li><a routerLink="second-component">Second</a></li>
  </ul>
</nav>

<router-outlet></router-outlet>
```

- Nicht file-based, sondern muss explizit definiert werden: *app.routes.ts*
- “First-match wins” Strategie
- Wildcard-Routes und dynamisches Routing mit Query-Parametern
- <router-outlet> Element definiert Stelle, an der geroutete Komponente angezeigt wird
- Verlinkung über *routerLink* Direktive, nicht href Attribut

Routing

other.component.ts

```
export class OtherComponent {  
  id: string;  
  
  constructor(  
    private route: ActivatedRoute,  
    private router: Router) {  
    this.id = this.route.snapshot.paramMap.get('id');  
  }  
  
  onHomeButtonClicked() {  
    this.router.navigate(['/']);  
  }  
}
```

- Programmatische Navigation über Route Service
- Abrufen von Query-Parametern über ActiveRoute Service

Aufgabe 4

20 Minuten

Rxjs für asynchrone Operationen

Recap: Promises

```
const myPromise = new Promise((resolve, reject) =>
  setTimeout(() => resolve("Hello World"), 3000);
);

myPromise.then(value => console.log(value));
```

Output:

1...

2...

3...

“Hello World!”

- Asynchrone Operation returned Ergebnis erst in der Zukunft (eventuell)
- Verknüpfung mit Callbacks für Erfolg- und Fehlerfall mittels *then* bzw. *catch*

Rxjs: Observables & Observer

```
const observer = {  
  next: x => console.log("New value: ", x),  
  error: err => console.log("Error: ", err),  
  complete: () => console.log('Done!')  
};  
  
const observable$ = new Observable((subscriber) => {  
  subscriber.next(1);  
  setTimeout(() => {  
    subscriber.next(2);  
    subscriber.complete();  
  }, 2000);  
});  
  
observable$.subscribe(observer);  
  
Output:  
“New value: 1”  
1...  
2...  
“New value: 2”  
“Done!”
```

Observable

- Asynchroner Datenstrom, der Werte emittieren kann
 - “Complete” wenn Datenstrom abgeschlossen ist
 - “Error” wenn ein Fehler auftritt

Observer

- “Beobachter”, der auf den Datenstrom reagiert bzw. diesen *subscribed*
 - *next* wenn neuer Wert emittiert wird
 - *error* wenn ein Fehler auftritt
 - *complete* wenn der Datenstrom endet

Rxjs: Observables & Observer

```
const observable$ = new Observable((subscriber) => {
  subscriber.next(1);
  setTimeout(() => {
    subscriber.next(2);
    subscriber.complete();
  }, 2000);
});

observable$.subscribe(x => console.log("New value: ", x));

Output:
"New value: 1"
1...
2...
"New value: 2"
```

- Observer kann auch unvollständig angegeben werden, wenn bspw. complete und error nicht interessieren
- Variablen für Observables werden per Konvention mit einem “\$” am Ende gekennzeichnet, um sie von “normalen” Objekten zu unterscheiden

Rxjs: Operatoren

```
const observable$ = new Observable((subscriber) => {
  subscriber.next(1);
  setTimeout(() => {
    subscriber.next(2);
    subscriber.complete();
  }, 2000);
});

const piped$ = observable$.pipe(
  tap(value => console.log('Original value:', value)),
  filter(value => value % 2 === 0),
  map(value => value * 10)
);

piped$.subscribe(x => console.log("New value: ", x));
```

Output:

“Original value: 1”

1...

2...

“Original value: 2”

“New value: 20”

- Emittierte Werte können mittels **pipe** durch verschiedene Operatoren
 - ...umgewandelt
 - ...mit Werten anderer Observables kombiniert
 - ...auf andere Observables abgebildet werden
 - und vieles mehr

Rxjs: Subscriptions

```
export class AppComponent implements OnDestroy {  
  data: any;  
  private subscription: Subscription;  
  
  constructor(private http: HttpClient) {  
    const dataStream$ = interval(5000).pipe(  
      switchMap(() => this.http.get("https://api.example.com/data"))  
    );  
  
    this.subscription = dataStream$.subscribe((response) => {  
      this.data = response;  
    });  
  }  
  
  ngOnDestroy() {  
    this.subscription.unsubscribe();  
  }  
}
```

- *subscriben* wir auf ein Observable, wird eine *subscription* returned
- Wenn die Daten nicht mehr benötigt werden, müssen wir die *subscription* beenden, um Memory Leaks zu vermeiden

Aufgabe 5

15 Minuten

Danke!