



Wer sind wir?

Hannes Frey

Anton Gerdts

Urs Weniger

<https://github.com/ursweniger/angular-workshop>

Agenda

Theorie und Setup

TypeScript

Angular Basics

Styling

Theorie und Setup

Was ist Angular?

- Framework zur Entwicklung von dynamischen SPAs
- Open-Source
- Entwickelt von Google
- Sprache: TypeScript

Vorteile von Angular

- Komponentenbasiert
- Zwei-Wege-Datenbindung
- Effektive Dependency Injection
- Komponentenbibliothek und umfangreiches Ökosystem
 - > Anwendungen mit komplexer Geschäftslogik und Verarbeitung von großen Datenmengen ermöglicht

Das Ökosystem

- Angular CLI
- Angular Material
- RxJS für asynchrone Calls
- NgRx für State-Management

Architektur

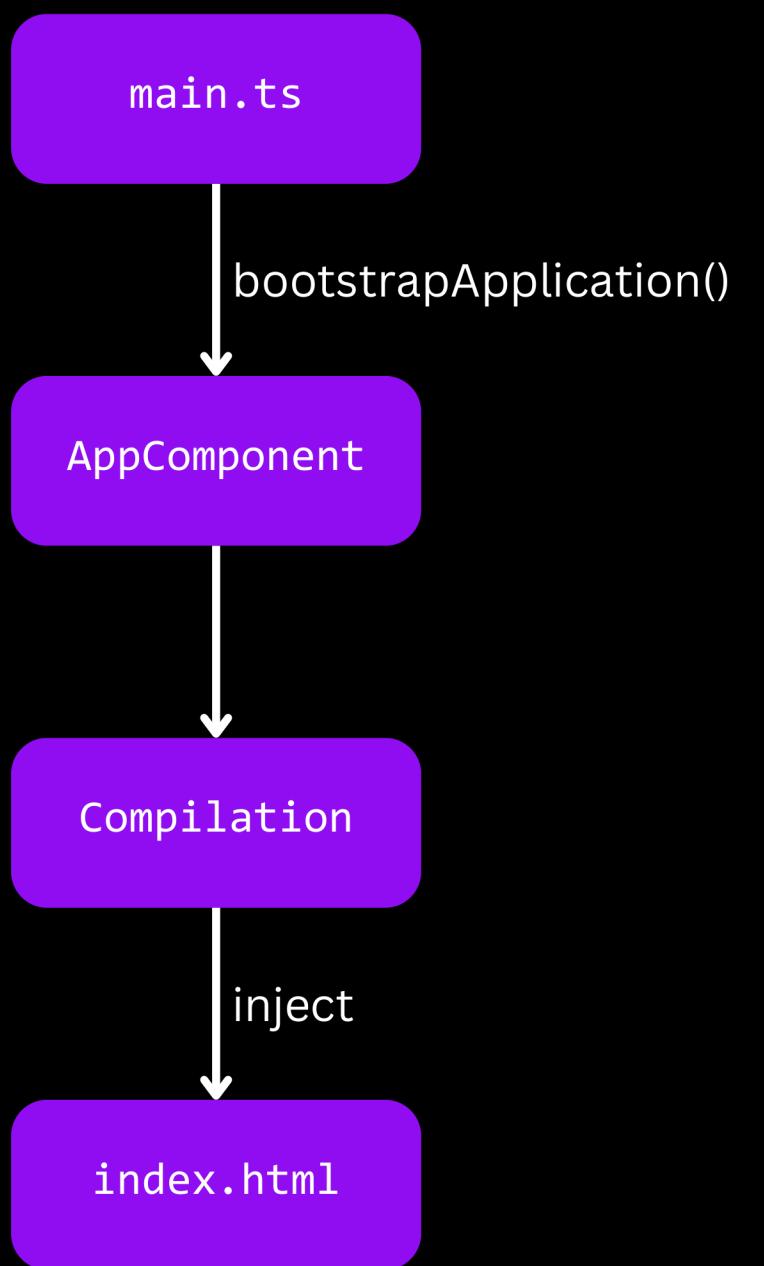
- Komponenten
- Templates
- Styles
- (Module)
- Services
- Dependency Injection

Projektstruktur

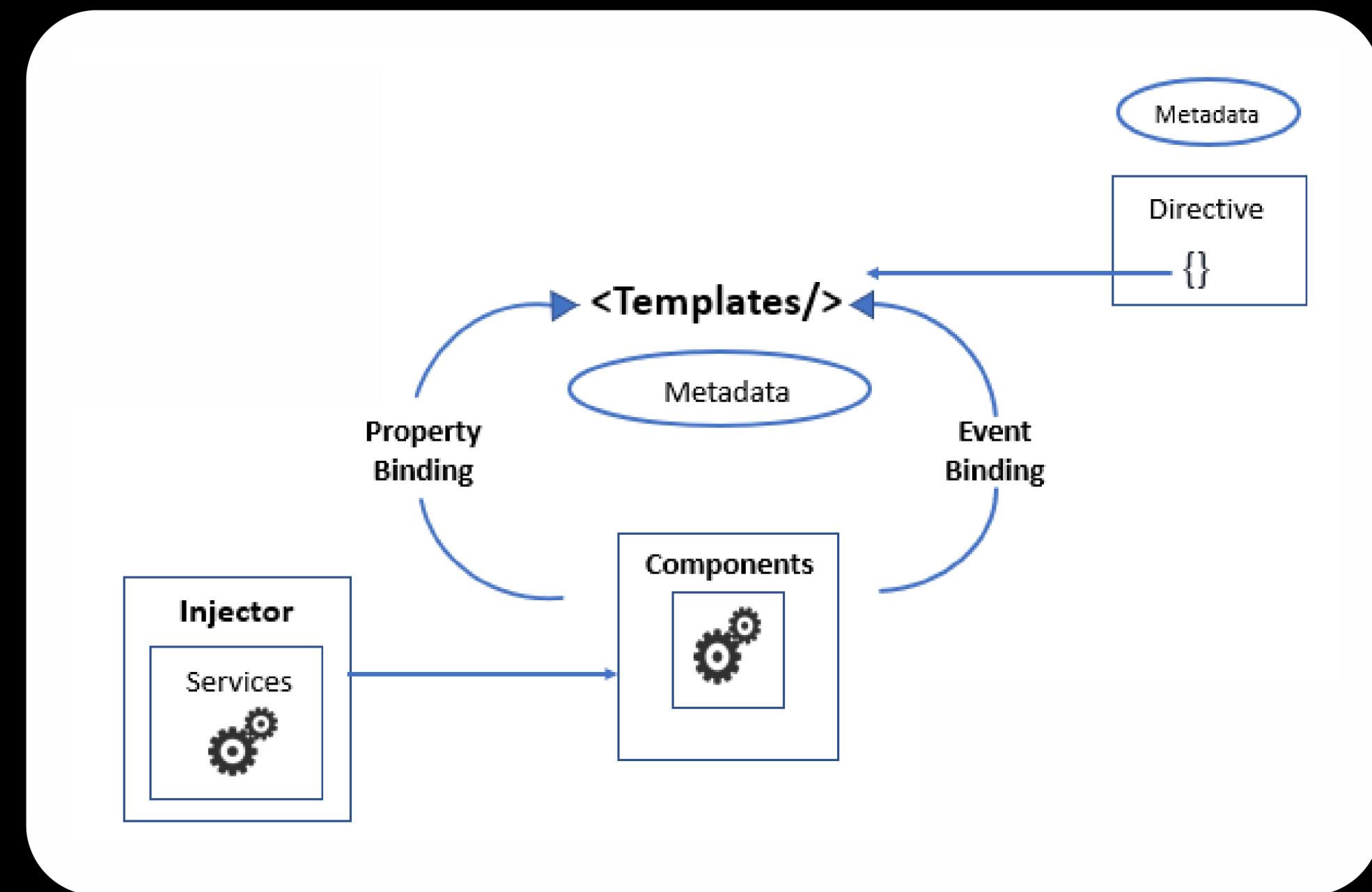
```
.  
├── public  
│   └── favicon.ico  
├── src  
│   ├── app  
│   │   ├── app.component.css  
│   │   ├── app.component.html  
│   │   ├── app.component.spec.ts  
│   │   ├── app.component.ts  
│   │   ├── app.config.ts  
│   │   └── app.routes.ts  
│   ├── index.html  
│   ├── main.ts  
│   └──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

Bootstrapping und Compilation

- Entrypoint: main.ts
- TypeScript Compilation zu JS
- Compilation von Angular Code zu plain HTML, CSS und JS
- Injektion des kompilierten Codes in index.html während Laufzeit



Angular zur Laufzeit



Installation und Setup

- Installation mittels Angular CLI
- Node Version >18.19
- lokales hosting auf localhost:4200

```
npm install -g @angular/cli
```

```
ng new my-app
```

```
cd my-app  
ng serve
```

Aufgabe 0

10 Minuten

TypeScript

Was ist TypeScript?

- Open-Source-Sprache, entwickelt von Microsoft
- Superset von JavaScript, das vornehmlich Typisierung ergänzt
- Transpiliert zu reinem JavaScript
- “Vorinstalliert” und Default in Angular

Vorteile von TypeScript

- Statische Typisierung
- Verbessertes Autocomplete in IDEs
- Frühzeitige Fehlererkennung
- Bessere Les- und Wartbarkeit des Codes

TypeScript Basics

- Typisierung mittels ":" und folgendem Typ

```
let isEasy: boolean = true;
```

- Häufige Typen:
boolean, number, string, object, void,
undefined, null, never

```
let answer: number = 42;  
let name: string = "Arthur";
```

- Sämtliche Typen erlauben: any

```
const notIntended: any = true;
```

- Union mit "|" erlaubt mehrere Typen,
Intersection mit "&" kombiniert Typen

```
let value = string | number;  
value = "answer";  
value = 42;
```

TypeScript Features

Arrays:

```
let myNumbers: number[] = [1, 2, 3];
let myNumbers: Array<number> = [1, 2, 3]
```

Assertions:

```
let someValue: any = "Hello there";
let strLength: number = (someValue as string).length;
```

Types and Interfaces:

```
type Person = {
  name: string;
  age: number;
};

interface Person {
  name: string;
  age: number;
};

const user: Person = {name: "Arthur", age: 42};
```

Tuples:

```
const: [string, number] = ["Arthur", 42];
```

Functions:

```
function add(a: number, b: number): number {
  return a + b;
}
```

Angular Basics

Komponenten

```
event-list-item
├── event-list-item.component.css
├── event-list-item.component.html
└── event-list-item.component.spec.ts
└── event-list-item.component.ts
```

- Bausteine der Benutzeroberfläche
- Bestehen aus:
 - Logik - name.component.ts
 - Markup - name.component.html
 - Styles - name.component.css
 - Tests - name.component.spec.ts
- generierbar durch CLI

Templates

app.component.ts

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

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

export class AppComponent {
```

app.component.html

```
<h1>Hello World!</h1>
```

app.component.ts

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

@Component({
  selector: 'app-root',
  standalone: true,
  template: `
    <h1>Hello World!</h1>
  `,
  styles: `CSS`,
})

export class AppComponent {
```

Komponenten

app.component.ts

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

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

export class AppComponent {
```

- Decorator für Konfiguration:
selector - Name für Imports
imports - Import für Abhängigkeiten
template/templateUrl - HTML
styles/styleUrl - CSS
- TypeScript Klasse für Logik:
State Management
User-Input handling
Methoden

Komponenten - Verwendung

event-list.component.ts

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

@Component({
  selector: 'app-event-list',
  standalone: true,
  imports: [EventListComponent],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class EventListComponent {
```

event-list-item.component.ts

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

@Component({
  selector: 'app-event-list-item',
  standalone: true,
  imports: [],
  templateUrl: './event-list-item.component.html',
  styleUrls: ['./event-list-item.component.css'],
})
export class EventListItemComponent {
```

event-list.component.html

```
<div>
  <app-event-list-item/>
</div>
```

Komponenten - State

event-list.component.ts

```
{...}  
export class EventListComponent{  
  title: string = 'Event List';  
  
  updateTitle() {  
    this.title = 'New Event List'  
  }  
}
```

- State wird in Klasse gesetzt
- Update von state in Klasse mit “this”

Komponenten - Eingaben

event-list.component.ts

```
{...}  
export class EventListComponent{  
  @Input() title?: string;  
}
```

- Eingabe von Daten in Komponente
- Setzen von state in Klasse mittels @Input()

app.component.html

```
<app-event-list title="Event List"/>
```

- Eingabe dann als Property der Komponente in aufrufendem Template

Data Binding

Interpolation

event-list.component.html

```
<h1>{{title}}</h1>
```

Property Binding

image.component.html

```
<img [src]="imageUrl">
```

Event Binding

button.component.html

```
<button (click)="handleClick()">Click Me!</button>
```

Two-Way Binding

app.component.html

```
<app-sizer [(size)]="fontSizePx"/>
```

Conditional Rendering

Basic Conditionals

event-list.component.html

```
@if (title) {  
  <h1>{{title}}</h1>  
} @else {  
  <h1>Fallback Title</h1>  
}
```

Loops

- Identifier muss mittels “track” gesetzt werden für spätere Updates

event-list.component.html

```
@for (snippet in textSnippets; track snippet.id) {  
  <p>{{snippet.text}}</p>  
}
```

Event handling

Event handler

button.component.html

```
<button (click)="handleClick()">Click Me!</button>
```

Event Object

button.component.html

```
<button (click)="passEvent($event)">Click Me!</button>
```

Event-Typen Beispiele

click

mouseover

focus

input

scroll

ngSubmit

Aufgabe 1

25 Minuten

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 {
```

```
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

Danke!