thecodecampus</>

Angular 2

Getting Started

TypeScript

Über mich

Philipp Burgmer

<> burgmer@w11k.de | @philippburgmer

Software-Entwickler, Trainer

Web-Technologien, Sicherheit

TypeScript, Angular </>

Über uns

theCodeCampus.de - Weiter.Entwickeln.

<> Schulungen seit 2007

Projekt-Kickoffs & Unterstützung im Projekt </>

w11k GmbH - the Web Engineers

<> Seit 2000

Auftrags-Entwicklung / Consulting

Web / Java

Esslingen / Siegburg </>

Überblick

<> Warum v2

<> Voraussetzungen

<> [Konzept + Beispiel]*

Probleme AngularJS 1

<> **Performance bei großen Anwendungen**
- Funktionsweise Data-Binding
- Datenfluss-Architektur

<> **Framework macht es zu leicht schlechten Code zu schreiben**
- Keine Struktur auf Code Ebene

<> **Konzeptionelle Probleme**
- Flacher Injektor (Silent-Override)
- Direktiven API
- Module nutzlos

Verbesserungen v2

<> **Weniger starres Framework**
- modularer (core, common, forms, routing, http, ...)
- mehr Teile auswechselbar (Strategy-Pattern)

<> **Moderen Konzepte umsetzten**
- Komponenten orientiertes Design
- Natives Modul-System

<> **Vereinheitlichtes Templating**

<> **Deutlich gesteigerte Performance**

<> **Verbessertes Tooling**

Neuerungen v2

<> Dynamisches Nachladen von Code

<> Server-Side-Prerendering (SEO, UX)

<> Alternative Renderer (NativeScript)

Vergleich zu React

<> **Äpfel mit Birnen**
- Framework statt Bibliothek für Rendering

<> **Weniger Performance vergleichen**

<> **Mehr High-Level-Architektur vergleichen**
- Was braucht man sonst noch?
- Wie spielt was zusammen?

Voraussetzungen

In AngularJS 1

<> Eine JS-Datei einbinden

<> `ng-app` an Tag

<> Fertig

In Angular 2

- <> ES6 Modul System
- <> Reactive Programming
- <> TypeScript
- <> Build-System

In AngularJS 1 - die Wahrheit

<> requireJS

<> ES6 oder TypeScript

<> Build-System & Module-Loader

Build-System

Build-System

<> **AngularJS 1**

- kein einheitliches Build-System
- großes Manko, erschwert den Einstieg sehr
- viele kleine

Abstraktionsniveau

<> Gulp, Grunt & Co sind nur Task-Runner, kein Build-Systeme

<> Build-System bauen komplexe Aufgabe

<> Viele Tasks, komplexes Zusammenspiel
- Wird schnell groß und unübersichtlich
- Wiederverwendbarkeit?

<> Situation vergleichbar mit Java vor 15 Jahren: ANT

Build-System

<> Angular CLI
- Vorgefertigter Build vom Angular Team
- Gibt Alternativen

<> **Aufgaben**
- `new` und `init`: Scaffolding
- `serve`: Development-Server
- `build`: Distribution bauen
- `generate`: Komponenten generieren
- `test`: Tests ausführen

TypeScript

<> **Ermöglicht Verwendung von modernen ES Features**
- Klassen
- Module
- Decorator

<> **Typen**
- Besseres Tooling
- Weniger Fehler erst zur Laufzeit
- Typisierung von Backend API möglich

<> **Kein Muss, aber sehr sinnvoll**

Reactive Programming

<> Welche Eigenschaften hat x?

```
1 var x = 1;
2 alert(x);
```

<> Welche Eigenschaften hat x?

```
1 var x = 1;
2 alert(x);
```

<> Sofort verfügbar (synchron)

<> Ein Wert

<> Welche Eigenschaften hat x?

```
1 var x = [1, 2, 3];
2 x.forEach(alert);
```

<> Sofort verfügbar (synchron)

<> Mehrere Werte (alle sofort verfügbar)

<> Welche Eigenschaften hat x?

```
1 var promise = async();
2 promise.then(function callback(x) {
3   alert(x);
4 });
```

<> Später verfügbar (asynchron)

<> Ein Wert (einmaliges Ergebnis)

Das Problem mit Promises

<> **Promise kann genau ein Mal aufgelöst werden**
- Gut für Asynchrone Ergebnisse
- Nicht geeignet für mehrere asynchrone Werte

<> **Lösung: Observable**
- Angular 2 verwendet nur Observables, keine Promises
- Implementierung: RxJS von Microsoft

<> Welche Eigenschaften hat x?

```
1 var observable = async();
2 observable.subscribe(function (x) {
3   alert(x);
4 });
```

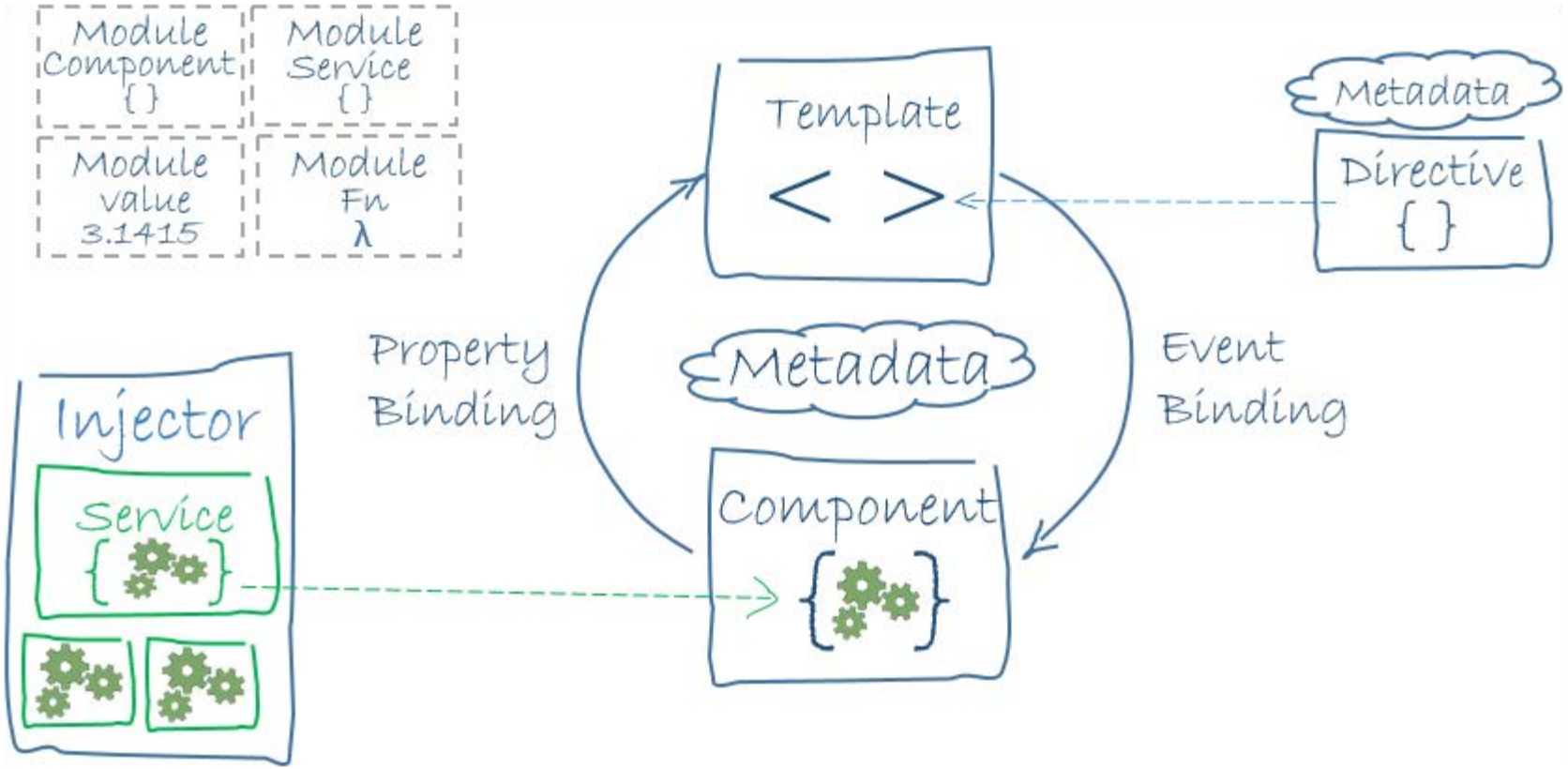
<> Später verfügbar (asynchron)

<> Mehrere Werte (verzöger verfügbar)

Konzepte

Komponenten

Architektur Überblick



Quelle: angular.io Dev-Guide

Komponenten

<> **Grundlegendes Konzept in Angular 2**

<> **Gesamtes UI ist aus Baum von Komponenten aufgebaut**

- "Anwendung" ist Top-Level-Komponente
- Komponenten können geschachtelt werden

<> **Komponenten bestehen aus**

- Template / View (HTML)
- Componenten Klasse
- Decorator an Klasse

Komponenten schreiben

<> **Komponenten werden im TypeScript Code definiert**

- Annotierte Klasse
- Klasse: Logik & Daten
- Annotation: Meta-Daten

```
1  import { Component } from "@angular/core";
2
3  @Component({
4    // css selector to find the component's usage
5    selector: "app-root",
6    template: `
7      <h1>Training App</h1>
8    `
9  })
10 export class AppComponent {}
```

Komponenten verwenden

<> **Komponenten werden im HTML verwendet**

- Verwendung muss genau Selektor entsprechen

```
1 <body>
2   <h1>Some static text not managed by Angular</h1>
3   <!-- matched by defined selector -->
4   <app-root></app-root>
5 </body>
```

Komponenten mit Data-Binding

<> Template nicht statisch sondern dynamisch

<> Instanz von Klasse stellt Daten bereit

<> Template zeigt Daten an
- Interpolation mit `{{ expression }}`
- Gibt Ergebnis der Expression als String aus

Komponenten mit Data-Binding

```
1  import { Training } from "./../training/training.model";
2  @Component({
3    selector: "app-root",
4    template: `
5      <h1>Training App</h1>
6      We offer {{trainings.length}} trainings
7    `
8  })
9  export class AppComponent {
10   public trainings: Training[];
11   constructor() {
12     this.trainings = [
13       new Training("Angular 2"), new Training("AngularJS 1"),
14       new Training("TypeScript"), new Training("Eclipse RCP")
15     ];
16   }
17 }
```

Direktiven

<> **Allgemeines Konzept**
- Komponenten
- Strukturelle Direktiven
- Attribut-Direktiven

<> ***Direktiven* meint meist strukturelle und Attribut-Direktiven**
- Wie Komponenten Teil des UI
- Unterschied zu Komponenten: kein eigenes Template

Strukturelle Direktiven

<> **Verändern DOM Struktur**
- Kein eigenes Template
- Kriegen Template übergeben

<> **Haben ein `*` als Prefix**

```
1 <ul>
2   <li *ngFor="let training of trainings">
3     {{training.name}}
4   </li>
5 </ul>
```

Strukturelle Direktiven in Angular 2 Common

<> `ngFor`
`ngIf`
`ngSwitch` </>

Module

Module

<> **Angular-spezifischer Code muss organisiert werden**

<> **Gleiche Grundgedanken und Aufbau wie bei ES6 Modulen**

- Kapselung & Wiederverwendbarkeit
- Leichtes Einbinden von Bibliotheken
- Genaue Steuerung was wo verwendet wird

<> **Kompromiss zwischen**

- "alles immer angeben müssen" (Angular 2 < 2.0.0-rc.5)
- "einfach rein schmeißen" (AngularJS 1)

Module

<> **Ein Modul ist eine annotierte Klasse**

- Klasse für gewöhnlich leer, nur Träger der Annotation
- Annotation enthält Informationen (Referenzen auf andere Module & Klassen)

<> **Angular-Module bestehen u.a. aus**

- Imports: Welche anderen Module werden intern verwendet
- Declarations: Was intern bekannt ist (quasi lokale Variablen)
- Exports: Was wird nach außen bekannt gemacht (für andere Imports)

```
1 @NgModule({
2   declarations: [ AppComponent ],
3   imports: [ BrowserModule, FormsModule, HttpModule ],
4   exports: [ AppComponent ]
5 })
6 export class AppModule { }
```

Bootstrapping

Bootstrapping

<> **Zusammenspiel Template & Code**
- `index.html`
- `main.ts` + imports

<> **Template-Teil in `index.html`**
- Angular 2 Anwendung besteht aus Baum von Komponenten
- Baum muss aufgebaut werden
- Tag für Root-Komponente in `index.html`

```
<body>
  <app-root></app-root>
</body>
```

Bootstrapping

<> **Angular 2 Anwendung muss von Hand per Code gestartet werden**
- Kein manuelles *dom ready* Handling notwendig

<> **Code-Teil in `main.ts`**
- Angular 2 stellt `bootstrapModule` Funktion bereit
- Parameter: Haupt-Modul der Anwendung

```
1 import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
2 import { AppModule } from './app/app.module';
3
4 platformBrowserDynamic().bootstrapModule(AppModule);
```

Zusammenhänge

<> **Ablauf bis hier her**

1. Aufruf `bootstrapModule` mit Modul (`AppModule`)
2. Modul ist Referenz auf eine Klasse
3. Klasse hat einen Decorator
4. Modul-Decorator enthält Komponenten

<> **Woher weis Angular welche Komponente für das Bootstrapping verwendet werden soll?**

Bootstrapping

<> Muss an Modul per `bootstrap` angegeben werden!

```
@NgModule({
  declarations: [ AppComponent ],
  imports: [ BrowserModule, FormsModule, HttpModule ],
  exports: [ AppComponent ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Zusammenhänge

<1> Aufruf `bootstrapModule` mit Modul (`AppModule`)

<2> Modul ist Referenz auf eine Klasse

<3> Klasse hat einen Decorator

<4> Modul-Decorator hat Feld `bootstrap`

<5> Feld `bootstrap` hält Referenz auf Komponenten Klasse (`AppComponent`)

<6> Komponenten-Klasse hat Decorator

<7> Komponenten-Decorator hat Feld `selector`

<8> Selector wird in DOM aus `index.html` gesucht

<9> Komponente wird instanziiert

Smarte vs. dumme Komponenten

<> **Dumme Komponenten**

- Sollen nichts über Umgebung wissen
- Bekommen Daten von außen übergeben
- Vergleiche Dependency Injection
- Sind dadurch wiederverwendbar

<> **Smarte Komponenten**

- Kennen ihre Umgebung
- Sind für Datenbeschaffung und Verarbeitung zuständig
- Geben Daten an dumme Komponenten weiter

Komponenten Inputs

<> **Komponenten können Daten von außen entgegennehmen**

- Eigenschaft in Klasse, in der die Daten gespeichert werden
- Eigenschaft wird mit `@Input()` markiert
- Wert kann in Template verwendet werden

```
1  import {Component, Input} from "@angular/core";
2
3  @Component({
4    selector: "training-details",
5    template: `<h3>{{training.name}}</h3>`
6  })
7  export class TrainingDetailsComponent {
8    @Input() public training: Training;
9  }
```

Komponenten Inputs

```
1 <training-details [training]="selectedTraining"></training-details>
```

<> **Daten können an Komponente übergeben werden**

- *Property Binding* über `[propertyName]="expression"`
- `[propertyName]` Property in der "inneren" Komponente *training-details*
- `expression` Property in der "äußeren / aufrufenden" Komponente

<> **Data-Binding in Angular 2 immer unidirektional**

- Objekt veränderbar
- Referenz bleibt außen immer gleich

Property Bindings

<> Werden für Inputs verwendet

<> Werden auch für Eigenschaften von DOM Elementen verwendet

<> Vereinheitlichtes Templating

```
1 <img [src]="training.image">
2 <button [disabled]="!training.discontinued"></button>
```

Event Bindings

<> **Data-Binding unidirektional**

- Daten von außen nach innen
- Wie kommen Informationen von innen nach außen?

Event Bindings

<> **Event Bindings werden auch für DOM-Events verwendet**
- DOM-Element wirft Event (Browser)
- Statement wird ausgeführt

<> **Keine extra Direktive notwendig**
- Vereinheitlichte Template-Sprache

```
1 <div class="star" (click)="likeTraining($event, training)">
2   <img src="favorite-star.png">
3 </div>
```

Komponenten Outputs

<> **Komponenten können Events nach außen werfen**

- Eigenschaft in Controller, vom Typ `EventEmitter`
- Typ-Parameter gibt Typ des Events an
- Eigenschaft wird mit `@Output()` markiert
- Events können über `EventEmitter#emit` gefeuert werden

```
1  import {Component, Input, Output, EventEmitter} from "@angular/core";
2  @Component({
3    selector: "training-details", template: `<h3>{{training.name}}</h3>`
4  })
5  export class TrainingDetailsComponent {
6    @Input() public training: Training;
7    // we use domain object as event, real world: ReleaseLikedEvent
8    @Output() public trainingLiked = new EventEmitter<Training>();
9    likeTraining() { this.trainingLiked.emit(this.training); }
10 }
```

Komponenten Outputs

<> **Mechanismus für UI spezifische Events**

- Benutzerinteraktion weiter geben
- Domain-Aktionen triggern
- Best Practice: Keine Business Logik in Componenten

```
1 <training-details [training]="training"
2                   (trainingLiked)="onTrainingLiked($event)">
3 </training-details>
```

Class- und Style-Bindings

<> Class-Bindings über `[class.className]="condition"`

```
<div [class.selected]="isSelected(training)"></div>
```

<> Style-Bindings über `[style.cssProperty]="value"`

```
<img [style.visibility]="training.image ? 'visible' : 'hidden'">
```

Services
&
Dependency Injection

Services

<> **Sehr allgemeines Konzept**

- Irgendwelche Funktionalität kapseln
- Nicht UI spezifisch
- Implementierungsdetails verstecken

<> **Einsatzmöglichkeiten**

- Höhere Abstraktion schaffen (REST)
- Integration anderer Bibliotheken (WebSocket)
- Datenhaltung, Datenzugriff, Caching, Business-Logik, ...

<> Services sollten in Angular 2 als Klassen implementiert werden

```
import { Training } from "./training.model";

export class TrainingService {
  getAll(): Observable<Training[]> {
    return Observable.from([]);
  }
}
```

Dependency Injection

<> **Allgemeines Konzept, in vielen Frameworks angewendet**

<> **Ziel: Lose Kopplung**

- Nicht an konkrete Implementierung binden
  - nur an Schnittstelle
  - keine direkte Instanziierung
- Nicht an konkrete Umgebung binden
  - keine Factories
  - keine Registries und Lookups

<> **Weg**

- Komponenten machen Abhängigkeiten nach außen sichtbar
- In Angular immer Konstruktor-Parameter
- Framework übernimmt Bereitstellung

Dependency Injection

<> **Verwendung bei Services**

- Injektion als Konstruktor-Parameter
- Service mit `@Injectable()` markieren
  - Heißt nicht, dass Service injezierbar
  - Heißt, dass Konstruktor Parameter injeziert werden sollen
  - Irreführender Name, in Java `@Inject` an Konstruktor

```
1 @Injectable()
2 export class ConsoleLogger extends Logger {
3   constructor(private console: Console) {}
4   debug() { this.console.debug("I do!"); }
5 }
```

Dependency Injection

<> **Verwendung bei Komponenten**

- Injektion als Konstruktor-Parameter
- Kein `@Injectable()` notwendig, da `@Component` verwendet

```
1 @Component({ selector: "training-details" })
2 export class TrainingDetailsComponent {
3   constructor(private log: Logger) {}
4   likeTraining() { this.log.debug(`Training ${this.training.name} liked`); }
5 }
```

Konfiguration

<> **Woher weiß Angular was zu injezieren ist?**

- Muss konfiguriert werden: *Providers*
- Injector -> Provider -> Instance

<> **Konfiguration an Modulen**

```
1 @NgModule({
2   providers: [ ] /* array of providers */
3 })
4 export class TrainingModule { }
```

<> **Wie sieht ein Provider aus?**

Konfiguration

<> **Shortcut**

- einfach nur die Klasse angeben: [ConsoleLogger]
- Shortcut für `[{ provide: ConsoleLogger, useClass: ConsoleLogger}]`

<> **Allgemein: Provider-Objekt**

- provide: Unter welchem Namen steht es zur Verfügung -> Klasse
- Ein Rezept, wie soll es erzeugt werden soll ->
  `useClass`, `useValue`, `useFactory` oder `useExisting`

```
1 @NgModule({
2   providers: [
3     { provide: ConsoleLogger, useClass: ConsoleLogger }, // same as shortcut
4     { provide: Logger, useExisting: ConsoleLogger } // different token, same instan
  ce
5   ]
6 })
7 export class TrainingModule { }
```

Hierarchische Injektoren

<> **Injektoren bilden einen Baum**
- Jede Komponente ein Child-Injector vom Injector der Vater-Komponente
- Instanzen sind Singletons in einem Injektor

<> **Provider für Child-Injektoren können an Komponenten definiert werden**
- Child-Injector kann Provider des Vaters überschreiben / ergänzen
- `providers` Property in `@Component` Decorator

<> **Einsatzmöglichkeiten**
- Auslösen von Namendskonflikten
- Unterschiedliche Konfiguration (z.B. Log-Level)

```
1 @Component({ providers: [ provide(Logger, { useClass: ConsoleLogger }) ]})
2 export class AppComponent {
3   constructor(private logger: Logger) { logger.level = 'warn'; }
4 }
5
6 @Component({ providers: [ provide(Logger, { useClass: ConsoleLogger }) ]})
7 export class TrainingDetailsComponent {
8   constructor(private logger: Logger) { logger.level = 'debug'; }
9 }
```

Formulare & Validierung

Model-Driven-Form

<> **2 Arten um Formulare aufzubauen**
- Template-Driven
- Model-Driven

<> **Model-Klassen um Formular zu beschreiben**
- `FormGroup` und `FormControl`
- Formular selbst ist eine `FormGroup`

<> `ReactiveFormsModule` **muss importiert werden**
- An dem Module, an dem Komponente deklariert wird

Model-Driven im Controller

<> Formular / Top-Level-Group als Member-Variable in Controller
- Zugriff in Controller und Template möglich

<> Initialisierung in *OnInit-Hook*, vorher kein Zugriff auf Daten (`@Input()`) möglich

```
1  import { FormGroup, FormControl } from "@angular/forms";
2  @Component({ })
3  export class TrainingFormComponent implements OnInit {
4    public form: FormGroup;
5    @Input() training: Training;
6
7    ngOnInit() {
8      this.form = new FormGroup({
9        name: new FormControl(this.training.name),
10       date: new FormControl(this.training.date)
11     });
12   }
13 }
```

Model-Driven im Template

<> `formGroup` **bindet Formular oder Gruppe**
- `form` auf oberster Ebene
- z.B. `div` oder `fieldset` innerhalb eines `form`

<> `formControl` **bindet Feld**
- Expression für Feld-Referenz

<> `formControlName` **bindet Feld aus nächst höherer Gruppe**
- String mit Feld-Name in nächst höherer Gruppe

```
<form [formGroup]="form" (ngSubmit)="onSubmit()">
  <label>Name:
    <input type="text" [formControl]="form.controls.name">
  </label>
  <label>Date:
    <input type="date" formControlName="date">
  </label>
</form>
```

Validierung

<> **Standard Validatoren**
- Validators.required
- Validators.minLength(minLength: number)
- Validators.maxLength(maxLength: number)
- Validators.pattern(pattern: string)

<> **Bisher keine Validierung anhand des Input-Type**

<> **Bereits Bibliotheken mit Validatoren verfügbar**

Validierung

<> Verwendung

```
import {FormControl, FormGroup, Validators} from "@angular/forms";

class PersonForm implements OnInit {
  ngOnInit() {
    this.form = new FormGroup({
      name: new FormControl('', Validators.required)),
      city: new FormControl('', Validators.maxLength(10)),
      zip: new FormControl('', Validators.pattern('[0-9]{5}')),
      street: new FormControl('', [
        Validators.required, Validators.minLength(3)
      ])
    });
  }
}
```

Fehler-Meldungen

<> **Validierung im Client**
- Schnelle Rückmeldung für den Benutzer
- Zwischen Fehlern und Hinweisen unterscheiden

<> **Zustand pro Feld / Gruppe / Formular**
- untouched, touched
- pristine, dirty
- valid, errors
- value

<> **Zustand von Gruppe ergibt sich aus Zustand enthaltener Felder**

Fehler-Meldungen

<> **Styling über automatisch gesetzte Klassen**

- Angular setzt CSS-Klassen an Formular und Feldern
- `ng-valid`, `ng-invalid`, `ng-pristine`, `ng-dirty`
- Styling über Klasse am Feld nur begrenzt möglich (CSS-Selektoren nur abwärts)

```
<style type="text/css">
  input.my-input.ng-invalid {
    border: 1px solid red;
  }
</style>
<input type="email" class="my-input" ngControl="email">
```

Fehler-Meldungen

<> **Zugriff auf Feld und Zustand**

- Erlaubt Dynamik an anderen Elementen

```
<form [formGroup]="form">
  <div class="form-group"
       [class.has-error]="form.controls.email.dirty && !form.controls.email.valid"
       [class.has-warning]="form.controls.email.pristine && !form.controls.email.valid">
    <input type="email" class="form-control" formControlName="email">
  </div>
</form>
```

Fehler-Meldungen

<> **Detailierte Meldungen pro Validator möglich**
- `errors`-Map mit Informationen pro Validator

<> **Code wird schnell unübersichtlich**
- Bibliothek mit Komponenten für Fehlermeldungen verwenden

```
<form [formGroup]="form">
  <input type="password" formControlName="password">
  <div [hidden]="!form.controls.password.dirty">
    <div [hidden]="!form.controls.password.errors?.required">
      Password is required
    </div>
    <div [hidden]="!form.controls.password.errors?.complex">
      Password is not complex enough
    </div>
  </div>
</form>
```

Routing

Routing / Navigation

<> Deep Linking über URL

<> Browser Historie (Vor / Zurück)

<> Pfad- oder Hash-Navigation

<> Hierarchischer Aufbau

<> Lazy Loading

<> Named Outlets

Routen

<> Route

- Relativer `path` (ohne führenden Slash)
- `component` für die Anzeige

```
1 import { Routes } from '@angular/router';
2
3 export const routes: Routes = [
4   { path: 'training', component: TrainingListComponent }
5 ];
```

<> Ablauf

- URL des Browsers wird überprüft / auf Änderungen überwacht
- Passende Route wird gesucht
- Komponente wird angezeigt

Default Route

<> **Was passiert wenn keine gültige Route aufgerufen wird?**

- Einstieg über URL ohne Route
- ungültige Route

<> **Fallback konfigurierbar**

- `path` mit `**`
- `redirectTo` Umleitung auf einen anderen Pfad
- Achtung: Reihenfolge der Routen spielt eine Rolle, Fallback als letzte

```
1 import { Routes } from '@angular/router';
2
3 export const routes: Routes = [
4   { path: 'training', component: TrainingListComponent },
5   { path: '**', redirectTo: '/training' }
6 ];
```

Router-Provider

<> Was passiert mit Routen-Konfiguration?

<> Wird verwendet um Provider für `Router` Service zu erzeugen

<> Provider wird von `RouterModule` bereitgestellt
- `RouterModule` muss in `AppModule` importiert werden
- Routing unterstützt Lazy-Loading
- Nicht alle Provider dürfen beim Nachladen registriert werden
- Unterscheidung `forRoot` und `forChild`

```
1 import { RouterModule } from '@angular/router';
2 import { routes } from './app.routes';
3
4 @NgModule({ imports: [
5   RouterModule.forRoot(routes)
6 ]})
7 export class AppModule { }
```

Komponenten im DOM

<> Wo wird Komponente der Route im DOM platziert?

<> Direktive `router-outlet` gibt Stelle zum Einfügen der Komponente vor

<> Router fügt dort die Komponenten, der aktuellen Route ein

```
@Component({
  selector: "app-root",
  template: `
    <h1>Training App</h1>
    <router-outlet></router-outlet>
  `
})
export class AppComponent {}
```

Konfiguration mit Parametern

<> **Problem:**

- Routen Konfiguration zur Entwicklungszeit, nicht zur Laufzeit
- Keine statische Route pro Datensatz möglich

<> **Lösung:**

- Dynamische Routen mit Parametern
- Routen Konfiguration kann in `path` Variablen enthalten

<> **Rest-Parameter vs URL-Parameter (?param=value)**

- Parameter in Route für Pflichtparameter
- URL-Parameter für optionale Parameter

```
1 {
2   path: 'training/:id',
3   component: TrainingDetailsComponent
4 }
```

Routen Parameter auslesen

<> `ActivatedRoute` **aus** `@angular/router` **injezierbar**

- `params` Feld mit Observable für Parameter
- Hintergrund: Komponenten-Instanz wird wiederverwendet

```
1  import { ActivatedRoute } from "@angular/router";
2  export class TrainingDetailsComponent implements OnInit, OnDestroy {
3    private subscription:   Subscription;
4
5    constructor(private route: ActivatedRoute) {}
6
7    ngOnInit() {
8      this.subscription = this.route.params
9        .subscribe(params => console.log(params['id']));
10   }
11
12   ngOnDestroy() { this.subscription.unsubscribe(); }
13 }
```

Routen Parameter auslesen

<> `Observable#map`: Wert auf anderer Wert

<> Was ist, wenn map ein `Observable<AndererWert>` zurück gibt?

<> Würde `Observalble<Observable<AndererWert>>` ergeben

<> `Observable#switchMap`

- Map + Switch
- Map gibt Observable zurück
- Switch: Nur zuletzt von map zurückgegebenes Observable weiter verwenden

```
1  export class TrainingDetailsComponent implements OnInit {
2    public training: Training;
3
4    ngOnInit() {
5      this.subscription = this.route.params
6        .map(params => +params['id'])
7        .switchMap(id => this.trainingService.getById(id))
8        .subscribe(training => this.training = training);
9    }
10 }
```

Navigation im Template

<> Direktive `routerLink` an a-Tag (erzeugt href, berücksichtigt LocationStrategy)
- `RouterModule` muss eingebunden werden damit verfügbar

<> Wert: Router-Link
- Array (für hierarchische Navigation)
- Namen und Parameter

<> Absolute und relative Links
- Absolute Links mit führendem Slash
- Zur aktuellen Route relative Links ohne führenden Slash

```
1 <a [routerLink]="['/training']">Trainings</a>
2 <a [routerLink]="['/training', training.id]">Some Training</a>
3 <a [routerLink]="[otherTraining.id]">See also</a>
```

Spec Tests

Spec-Tests

<> a.k.a Unit-Tests

<> Tools

- Karma als Test-Runner
- Jasmine als Test-Framework (andere möglich, Mocha, QUnit)
- `@angular/core/testing` und `@angular/http/testing` für Test-Support
- Istanbul für Code-Coverage

Angular - inject

<> Angular stellt `inject` Funktion bereit (`@angular/core/testing`)

<> Wrapped an `it` übergebene Test-Funktion

- suboptimale API, TypeScript Typen werden nicht genutzt
- 1. Parameter: Was soll injiziert werden
- 2. Parameter: Funktion, die mit DI aufgerufen wird

```
1 it(
2   'should ...',
3   inject([Service], (service: Service) => {
4     expect(service).toBeDefined();
5   })
6 );
```

Angular - Provider

<> Provider werden in `beforeEach` Funktion konfiguriert

<> Angular stellt `TestBed` Klasse zur Verfügung
- `configureTestingModule` um Module mit Provider für Test zu erzeugen

<> Module und somit auch Injector wird für jeden Test neu erzeugt

```
1  beforeEach(() => {
2    this.mockedDependency = {
3      aMethod: jasmine.createSpy('aMethod').and.callFake(() => {})
4    };
5    TestBed.configureTestingModule({
6      providers: [
7       ServiceWithDependency,
8       { provide: Dependency, useValue: this.mockedDependency }
9      ]
10   });
11 });
```

Angular - async

<> Umgang mit Jasmine-Done-Funktion umständlich & fehleranfällig

<> Angular verwendet intern *zone.js* -> weis was wann ausgeführt wird

<> `async` Funktion aus `@angular/core/testing`

- Wrapped an `it` übergebene Test-Funktion (inklusive `inject`)
- Ruft `done` automatisch auf wenn alle asynchronen Aufrufe abgearbeitet

```
1 it('should call promise handler', async(function () {
2   someAsyncCall().then(function (result) {
3     expect(result).toBeDefined();
4   }));
5 });
```

Http testen

<> **Unit-Tests sollten keine HTTP Request abschicken**

- Langsam
- Setzt Backend voraus

<> **Wie Abhängigkeit auf `Http` mocken?**

- Schnittstelle zu groß
- Zu viele Parameter
- Nicht `Http` mocken
- Nur Verbindungen mocken

Http MockBackend

<> `Http` **erstellt Verbindungen nicht selbst**
- Abstrakte Klasse `ConnectionBackend`
- Implementierungen: `XHRBackend` und `MockBackend`

<> **Für Test Provider überschreiben**
- Provider `MockBackend` -> MockBackend injizierbar (extra API für Test)
- `ConnectionBackend` mit `useExisting: MockBackend` -> gleiche Instanz

```
1  import { XHRBackend, ConnectionBackend, HttpModule } from '@angular/http';
2  import { MockBackend } from '@angular/http/testing';
3
4  describe('Service: Training', () => { beforeEach(() => {
5    TestBed.configureTestingModule({
6      imports: [ HttpModule ],
7      providers: [
8        MockBackend,
9        { provide: ConnectionBackend, useExisting: MockBackend }
10     ]
11   });
12 }); });
```

Http MockBackend

<> `MockBackend` **in Test injizieren lassen**

<> `MockBackend` **stellt Observable für Verbindungen bereit**

- Wenn Verbindung angefordert wird, wird Subscriber aufgerufen
- Subscriber kann Antwort liefern

```
it('should get trainings',
  async(inject([MockBackend, TrainingService], (mockBackend, trainingService) => {
    mockBackend.connections.subscribe((connection: MockConnection) => {
      connection.mockRespond(new Response(
        new ResponseOptions({ body: [{ id: 0, name: "Angular 2" }]})
      ));
    });

    trainingService.getAll().subscribe(trainings => {
      expect(trainings.length).toBe(1);
      expect(trainings[0].id).toBe(0);
    });
  }))
);
```

Komponenten testen

<> **Service, Pipe, Guard, ...**

- Bestehen nur aus Code
- Plain Jasmine, manuelles Instanziieren
- Jasmine + Angular DI
- Klassische Unit-Tests

<> **Component**

- zwei Ebenen: nur Code oder Code und Template
- nur Code: wie oben
- Code und Template: DOM im Test?

TestBed

<> `TestBed` hilft bei Instanziierung

<> Komponente über `createComponent` erzeugen lassen

```
1 it('should render the training name', async(() => {
2   let fixture = TestBed.createComponent(TrainingDetails);
3 });
```

<> Liefert `Fixture`

```
1 abstract class ComponentFixture {
2   debugElement;       // test helper
3   componentInstance;  // access properties and methods
4   nativeElement;      // access DOM
5   detectChanges();    // trigger component change detection
6 }
```

TestBed

```
it('should render the training name', async(() => {
  let fixture = TestBed.createComponent(TrainingDetailsComponent);

  let component = fixture.componentInstance;
  let element = fixture.nativeElement;

  component.training = { name: 'Angular 2' };
  fixture.detectChanges();

  expect(element.querySelector('h3').innerText).toBe('Angular 2');
});
```

<> **Komponenten**
- Lifecycle Hooks
- Interaktion mit Kindern
- Style Encapsulation

<> **Pipes**

<> **Routing Guards**

Fazit

<> **Steile Lernkurve für Frischlinge**

<> **Vieles aus AngularJS 1 bekannt**
- Data-Binding, Services, DI, Pipes
- Leichter Umstieg von AngularJS 1
- Aber auch einiges neu (RxJS, TypeScript)

<> **Bereits viele Bibliotheken, von Community schnell aufgegriffen**
- Styling, Komponenten, Datenfluss

<> **Nach Einarbeitung: runde Sache, macht Spaß, fühlt sich gut an**

Philipp Burgmer
burgmer@w11k.de
Twitter: @philippburgmer
GitHub: pburgmer

www.thecodecampus.de
@theCodeCampus