












Blockwoche: Web Programming Lab

Programm Blockwoche

Montag 	Dienstag  	Mittwoch  	Donnerstag   	Freitag   
Architekturansätze von Web Anwendungen JavaScript Sprachkonzepte I	Client-Side- JavaScript I	Angular	Angular	Progressive Web Apps
JavaScript Sprachkonzepte II	Client-Side- JavaScript II	Angular	Server-Side- JavaScript REST	Authentication @ Web Apps

Agenda

1. Organisatorisches
2. Besprechung Http Übung
3. Angular Forms
4. Angular Testing
5. RxJS

Input

Angular Forms



Angular Forms

- Angular bietet zwei unterschiedliche Ansätze zum Erstellen von Formularen:
 - **Template Driven Forms**
 - **Reactive Forms (dynamisch)**

Template Driven Forms

- Die Form sowie Validierungen werden im HTML definiert.
- Die Verknüpfungen zur Component werden mit verschiedenen Direktiven ermöglicht:
 - ngForm
 - ngModel
- Form-spezifische Funktionen sind im "FormsModule" enthalten.

Template Driven Forms

```
<form #itemForm="ngForm" (ngSubmit)="onSubmit(itemForm)">
  <label>Name</label>
  <input class="form-control" name="name" required [(ngModel)]="model.name" />
  <button type="submit">Submit</button>
</form>
<div [hidden]="!itemForm.form.valid">
  <p>Alles ok!</p>
</div>
```

Template Driven Forms

- Demo

Template Variables

- Template Variablen helfen, Daten aus einem Teil des Templates in einem anderen Teil des Templates zu nutzen.
- Werden meist verwendet, um Formulare flexibler zu gestalten.

Übung Forms 1: Template Driven Form

- Erstelle ein Kontakt-Formular in deiner Heroes App :
 - Erstelle dafür eine neue Komponente inkl. Link auf dem Dashboard
 - Erstelle ein Feld "Nachricht" sowie "Absender E-Mail"
 - Füge Validierungen für das Pflichtfeld "Nachricht" sowie die E-Mail Adresse ein mit entsprechenden Fehlermeldungen.
 - Der Submit des Formulars soll nicht möglich sein, wenn Validierungsfehler enthalten sind.
 - Logge bei einem validen Submit die Daten auf der Konsole.

Reactive Forms

- Definition der Validierungen in der Component
- Beim reaktiven Ansatz wird die Validierungslogik aus dem Template entfernt, wodurch die Templates übersichtlicher bleiben.
- Erzeugung von Formular-Elementen in der Component
- Funktionen sind im “ReactiveFormsModule” enthalten

Reactive Forms

- Demo

Übung Forms 2: Reactive Form

- Erstelle aus der Template Driven Form von "Übung Forms 1" eine Reactive Form.

Angular Material

Angular Material

Material Design components for Angular

Get started

High quality

Internationalized and accessible components for everyone. Well tested to ensure performance and reliability.

Straightforward APIs with consistent cross platform behaviour.

Versatile

Provide tools that help developers build their own custom components with common interaction patterns.

Customizable within the bounds of the Material Design specification.

Frictionless

Built by the Angular team to integrate seamlessly with Angular.

Start from scratch or drop into your existing applications.

Zusatz-Übung: Angular Tour of Heroes

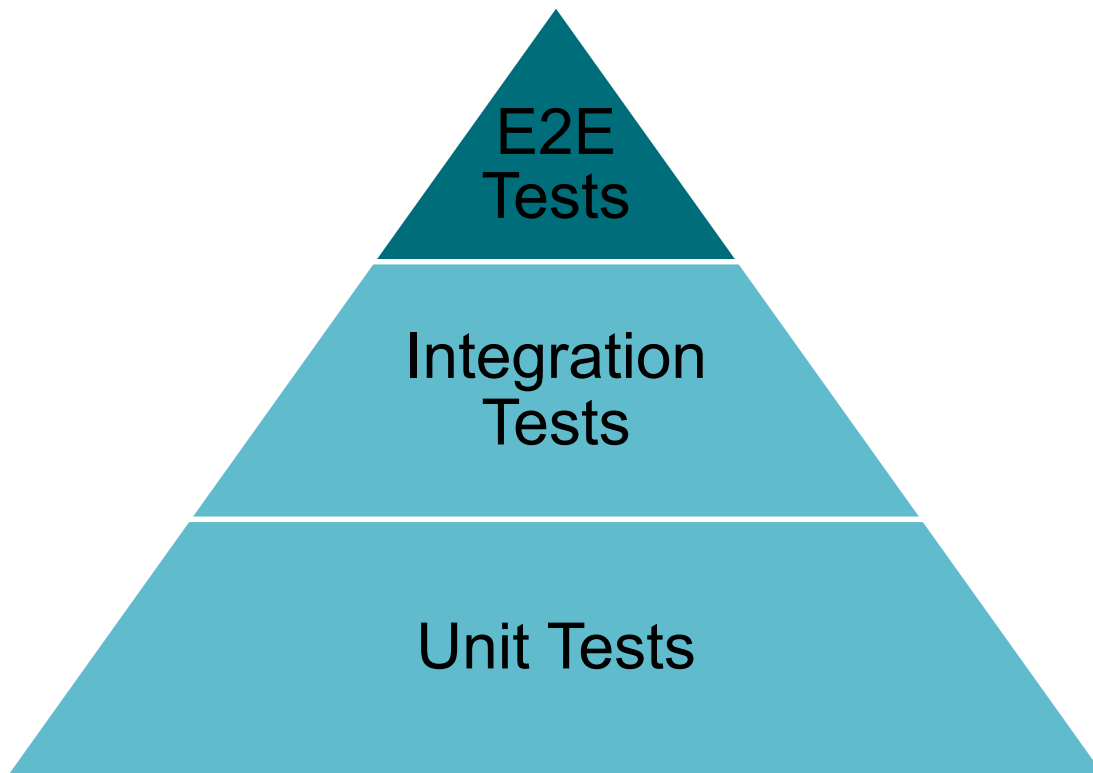
- Integriere Angular-Material in deine Tour of Heroes App!
- <https://material.angular.io/>

Input

Angular Testing



Angular Testing



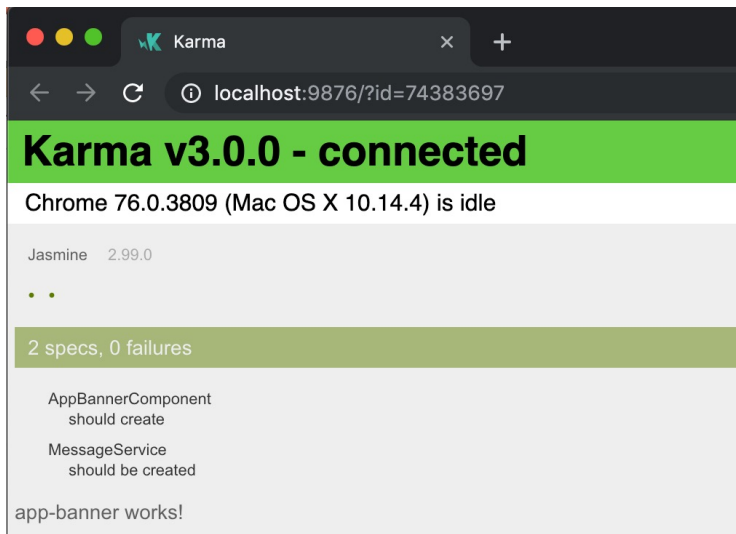
Angular Testing

- Out of the Box via Angular CLI
- Unit Testing mit Karma und Jasmine
(auch via Jest oder Cypress möglich)
- End-to-End Testing mit Cypress
- Test-Stubs werden automatisch generiert (TDD ready)
- Integrationsmöglichkeiten in CI Build (Headless Browser)
- Code Coverage Reports

Angular Testing

```
ng test
```

```
Karma v3.0.0 server started at http://0.0.0.0:9876/  
Launching browser Chrome with unlimited concurrency  
Starting browser Chrome  
Executed 2 of 2 SUCCESS (0.077 secs / 0.069 secs)  
TOTAL: 2 SUCCESS
```



Angular Testing: Jasmine

- Behaviour-Driven-Development (BDD) Framework zum Testen von JavaScript Code.

```
describe("A suite is just a function", () => {  
  let a;  
  
  it("and so is a spec", () => {  
    a = true;  
  
    expect(a).toBe(true);  
  });  
});
```

Angular Testing: Service Test

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

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

  constructor(private http:HttpClient) { }

  getMessages(): Observable<any> {
    return this.http.get('api/messages.json');
  }
}
```

Angular Testing: Service Test

```
describe('MessageService', () => {  
  
  let httpClientSpy: { get: jasmine.Spy };  
  let messageService: MessageService;  
  
  beforeEach(() => {  
    httpClientSpy = jasmine.createSpyObj('HttpClient', ['get']);  
    messageService = new MessageService(<any>httpClientSpy);  
  });  
  
  it('should get messages', () => {  
    const expectedMessages: any[] = [{ id: 1, name: 'A' }, { id: 2, name: 'B' }];  
  
    httpClientSpy.get.and.returnValue(of(expectedMessages));  
  
    messageService.getMessages().subscribe(messages =>  
      expect(messages).toEqual(expectedMessages, 'expected messages'), fail);  
  
    expect(httpClientSpy.get.calls.count()).toBe(1, 'one call');  
  });  
});
```

Angular Testing: Service Test

- Demo

Übung Testing 1: Service Test

- Erstelle Unit Tests für den Service **hero.service.ts**
 - Funktion **getHeroes**
 - Funktion **getHero**
 - Funktion **addHero**
- Zusatz: Messe die Code-Coverage des Tests

Angular Testing: TestBed

- Nutze TestBed, um die entsprechenden Abhängigkeiten zu laden, damit sie während der Tests verfügbar sind.

```
beforeEach(() => {  
    TestBed.configureTestingModule({  
        providers:[ValueService, DialogService]  
    });  
});
```

```
it('should use ValueService', () => {  
    service = TestBed.get(ValueService);  
    expect(service.getValue()).toBe('real value');  
});
```

Angular Testing: Mocken von Abhängigkeiten

- Nutze Spies von Jasmine, um einzelne Units isoliert von Abhängigkeiten zu anderen Klassen zu testen.

```
beforeEach(() => {  
  TestBed.configureTestingModule({ providers: [  
    ValueService,  
    { provide: DialogService, useValue: jasmine.createSpyObj('dialogService',  
      ['openSnackBar']) }  
  ] });  
});
```

```
it('should use ValueService', () => {  
  let service = TestBed.get(ValueService);  
  expect(service.getValue()).toBe('real value');  
  
  let dialogService = TestBed.get(DialogService);  
  expect(dialogService.openSnackBar).toHaveBeenCalled();  
});
```

Angular Testing: Component Test

```
@Component({
  selector: 'app-app-banner',
  templateUrl: './app-banner.component.html',
  styleUrls: ['./app-banner.component.css']
})
export class AppBannerComponent implements OnInit {

  constructor(private messageService: MessageService) { }

  ngOnInit() {
  }

  getMessages(): any[] {
    return this.messageService.getMessages();
  }

}
```

Angular Testing: Component Test

```
describe('AppBannerComponent', () => {
  let component: AppBannerComponent;
  let fixture: ComponentFixture<AppBannerComponent>;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ AppBannerComponent ],
      providers: [{ provide: MessageService, useValue: jasmine.createSpyObj('messageService',
        ['getMessages']) }]
    }).compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(AppBannerComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should get messages', () => {
    TestBed.get(MessageService).getMessages.and.returnValue([]);
    expect(component.getMessages()).toEqual([]);
  });
});
```

Übung Testing 2: Component Test

- Erstelle Unit Tests für die Komponente **heroes.component.ts**
 - Funktion **getHeroes**
 - Funktion **add**
 - Funktion **delete**
- Mocke Abhängigkeiten via Spies
- Nutze das TestBed von Angular

Angular E2E Test: App under Test

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

<div>
  Points: <span>{{points}}</span>
</div>

<button (click)="plus1()">Plus 1</button>
<button (click)="reset()">Reset</button>
```

```
@Component({
  selector: 'app',
  templateUrl: './app.component.html',
  styleUrls: [ './app.component.css' ]
})
export class AppComponent {
  title = 'Letslearn';
  points = 1;

  plus1() {
    this.points++;
  }

  reset() {
    this.points = 0;
  }
}
```

Cypress E2E Test: Test Spec's

```
describe('App Tests', () => {  
  it('Should start with 1 point', () => {  
    cy.visit('/');  
    getPointsElement().should('have.text', '1');  
  });  
  
  function getPointsElement() {  
    return cy.get('div').contains('Points').get('span');  
  }  
});
```

Angular E2E Test: Test Spec's

- Demo



- Angular Doc's: <https://angular.io/docs>
- Angular Resources: <https://angular.io/resources>



Input

RxJS



Reactive Programming

- **Reactive Programming:** Programmierung mit asynchronen Datenströmen. Ermöglicht einfacheres Handling mit asynchronem Code.
- **ReactiveX:** API für Reactive Programming in verschiedenen Plattformen:
 - JavaScript: **RxJS**
 - Java: **RxJava**
 - C#: **Rx.NET**

Observables

- Ein **Observable** repräsentiert eine Menge von Daten, die in einer noch nicht bekannten Menge zu einem noch nicht bekannten Zeitpunkt bereitstehen werden.
- Ein **Observer** abonniert ein Observable (via «subscribe») und damit die später ankommenden Daten. Ein Observable kann gefiltert, gruppiert oder transformiert werden.
- Ein Observable hat einen **Zustand**. Solange es nicht abonniert wurde, ist es «**cold**». Die Werte werden nicht verarbeitet. Erst durch eine Subscription wird ein Observable «**hot**», die Werte werden dadurch überwacht und verarbeitet.

Observables vs. Promises

- Beides sind Platzhalter für noch nicht bekannte Ergebnisse.
- Observables können abgebrochen werden, Promises nicht.
- Observables können mehrere Ergebnisse verarbeiten, Promises nur ein Ergebnis.

Let's dive into code....

Setup

- Editor wie z.B. Visual Studio Code oder IntelliJ
- NodeJS
- Node Package Manager (npm)
- Chrome
- git

→ `git clone https://github.com/web-programming-lab/rxjs-seed`

→ `cd ./rxjs-seed && npm install`

→ `npm run start`

Crate Observable

```
import {Observable, Observer} from 'rxjs';

const observable = new Observable((observer: Observer<any>) => {
  observer.next('Hello');
  observer.next('World');
  observer.complete();
});

observable.subscribe((value) => console.log(value));
```

Create Observable

```
import {Observable, Observer} from 'rxjs';

const observable = new Observable((observer: Observer<any>) => {
  observer.next('1');
  observer.next('2');
  observer.next('3');
  observer.complete();
  observer.next('4');
});

observable.subscribe((value) => console.log(value));
observable.subscribe((value) => console.log(value));
```

Erstellung von Observables aus einem Arrays

```
import { from } from 'rxjs';  
  
from([1, 2, 3])  
  .subscribe((value) => console.log(value));
```

Console



☒ Clear console on reload

1

2

3

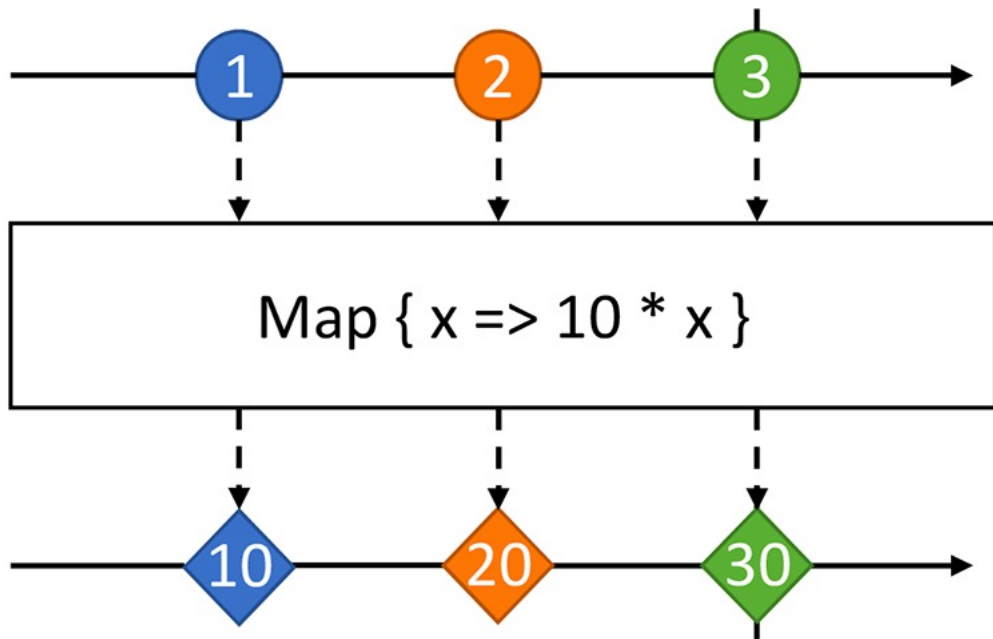
>





Operatoren

- Toolbox, um Werte eines Observables ...
 - zu verändern
 - mit anderen Observables zu vereinen
 - zu filtern
 - zu limitieren
 - usw.

Operatoren: Marble Diagramme

- Veranschaulichung eines Observables über die Zeit.

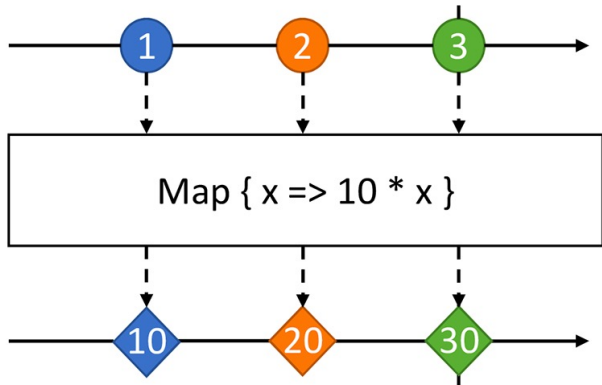


Symbol	Bedeutung
	Einzelner ausgestoßener Wert
	Transformierter Wert in der gleichen Farbe des Quellwertes
	Terminierung der Observable-Sequenz durch einen Fehler (im Beispiel nicht verwendet)
	Normale Terminierung der Observable-Sequenz (complete)

Operatoren: Transformation von Observables

- Map:** Transformiert den eingehenden Wert in einen beliebigen anderen Wert.

```
import { from } from 'rxjs';  
import { map } from 'rxjs/operators';  
  
from([1, 2, 3])  
  .pipe(map((x) => x * 10))  
  .subscribe((value) => console.log(value));
```



Console



☒ Clear console on reload

10

20

30

>

Operatoren: Filtern von Observables

- **Filter:** Begrenzt die Elemente einer Verarbeitungskette anhand einer Funktion, welche für jeden Wert einen Wahrheitswert zurückgibt.

```
import {from} from 'rxjs';  
import { filter } from 'rxjs/operators';  
  
from(['Jim', 'Patrick', 'Andreas', 'Johnny', 'Jack'])  
  .pipe(filter((value) => value.startsWith('J')))  
  .subscribe((value) => console.log(value));
```

Console



☒ Clear console on reload

Jim

Johnny

Jack



RxJS Übung 1: Filter

- Filtert die Studenten, so dass in der Result Liste nur noch Studenten der Klasse «1» enthalten sind.
- Branch: students

RxJS Übung 2: Sum

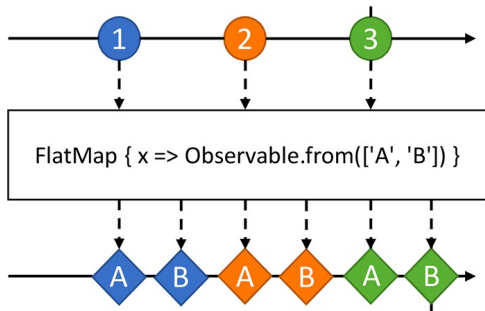
- Summiere alle Nummern in der Liste und gib die Summe auf der Konsole aus.
- Tipp: <https://www.learnrxjs.io/learn-rxjs/operators/transformation/scan>
- Branch: sum

Operatoren: Transformation von Observables

- **FlatMap**: Transformiert einen einfachen Eingangswert in ein Observable, dessen Sequenzwerte anschließend jeweils einzeln in der Sequenz weiterverarbeitet werden.

```
import { from } from 'rxjs';
import { flatMap } from 'rxjs/operators';

from([1, 2, 3])
  .pipe(flatMap((value) => from(['A', 'B'])))
  .subscribe((value) => console.log(value));
```



Console



☒ Clear console on reload

A

B

A

B

A

B



Operatoren: Take

- **Take:** Begrenzt die verarbeiteten Werte auf eine übergebene Anzahl.

```
import { interval } from 'rxjs';
import { take } from 'rxjs/operators';

interval(100)
  .pipe(take(5))
  .subscribe((value) => console.log(value));
```

Console



☒ Clear console on reload

0

1

2

3

4

> |

Operatoren: Kombination von Observables

- **Zip**: Kombiniert die Werte von mehreren Observables 1:1 miteinander.

```
import { from, zip } from 'rxjs';
import { map } from 'rxjs/operators';

const nameObservable = from(['Jim', 'Johnny', 'Jack']);
const surnameObservable = from(['Beam', 'Walker', 'Daniels', 'Jameson']);

zip(nameObservable, surnameObservable)
  .pipe(map((values) => `${values[0]} ${values[1]}`))
  .subscribe((value) => console.log(value));
```

Console



☒ Clear console on reload

Jim Beam

Johnny Walker

Jack Daniels



Operatoren

- Es gibt über 90 weitere Operators
- Weitere Operatoren: <https://www.learnrxjs.io/operators/>