












# **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

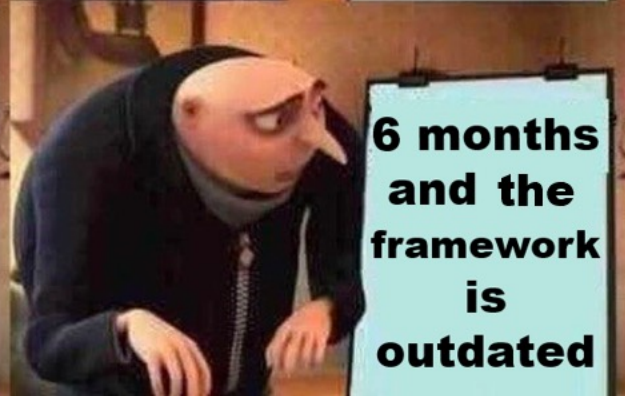
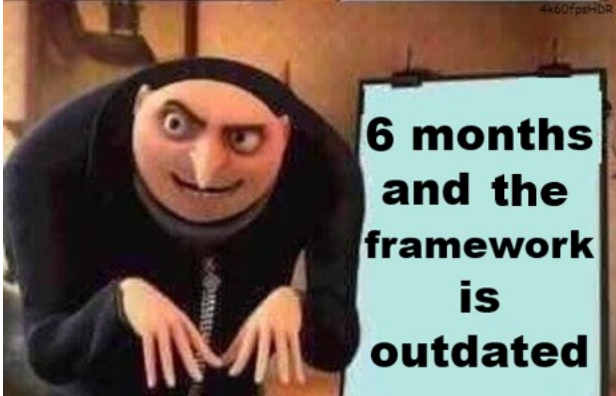
# Agenda

1. CSS & JavaScript Libraries
2. Single Page Application (SPA)
3. Angular Architecture
4. Angular Components
5. Angular Directives
6. Angular Services & Dependency Injection
7. Angular Pipes
8. Angular Routing
9. Angular HTTP

# Input

## CSS & JavaScript Libraries





# CSS Frameworks

- **Bootstrap:**
  - Responsive Grid System
  - Ready to use Komponenten (Buttons, Navbar, Carousel etc.)
  - Plugins (auf Basis von jQuery)
- **Bulma:**
  - Responsive
  - Ready to use Komponenten (Buttons, Navbar, Carousel etc.)
  - Modular
  - Basierend auf Flexbox
- **Tailwindcss**
  - Responsive
  - Utility-first

# JavaScript Libraries & Frameworks

- **NO Library:** DOM API (<http://youmightnotneedjquery.com/>)
- **Libraries:**
  - jQuery
  - Umbrella JS
  - nanoJS
  - ...
- **Frameworks:**
  - Angular
  - React
  - Vue.js
  - ...

# Javascript Frameworks

● React  
Programm

● Angular  
Software

● Vue.js  
Thema

+ Vergleich hinzufügen

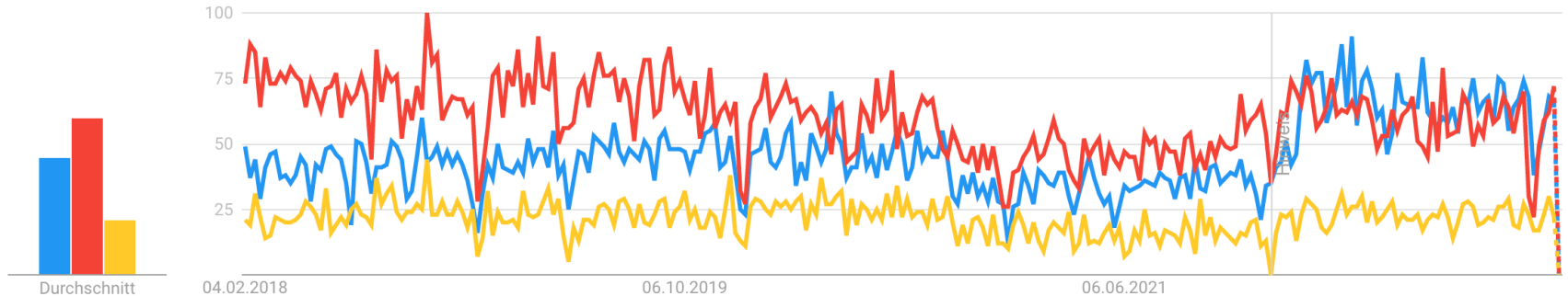
Schweiz ▼

Letzte 5 Jahre ▼

Alle Kategorien ▼

Websuche ▼

Interesse im zeitlichen Verlauf ?





# Input

## Single Page Application (SPA)



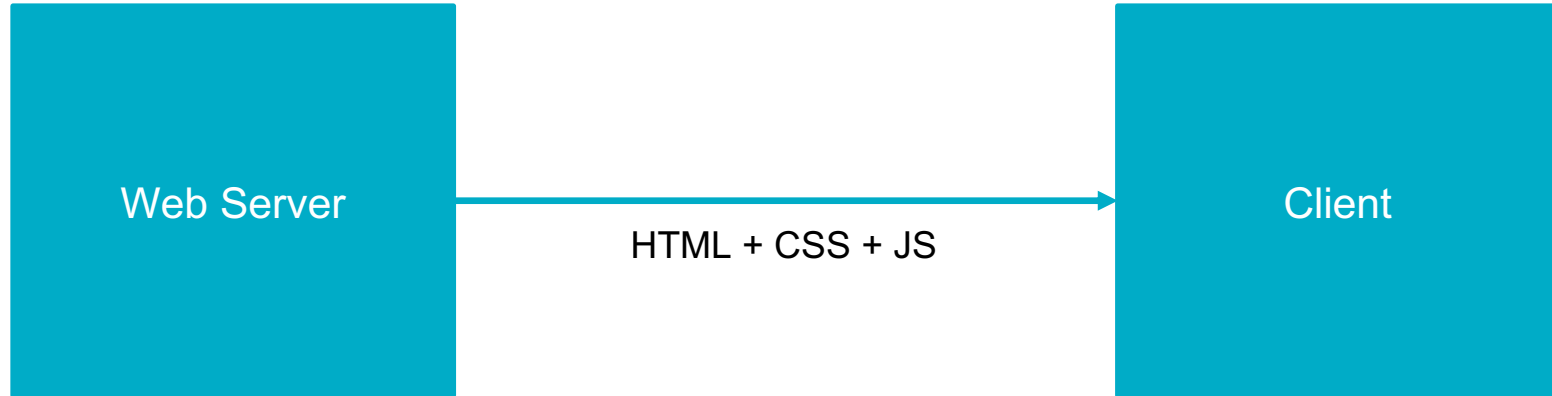
# SPA – Evolution

- 1990 – HTML
- 1993 – CGI
- 1995 – JavaScript, PHP
- 1996 – ASP
- 1998 – CSS2
- 1999 – JSP
- 2002 – ASP.NET
- 2004 – Ruby on Rails
- 2006 – jQuery
- 2008 – Draft HTML5
- 2009 – AngularJS
- 2010 – KnockoutJS / Backbone.JS
- 2016 – Angular / React / Vue.js

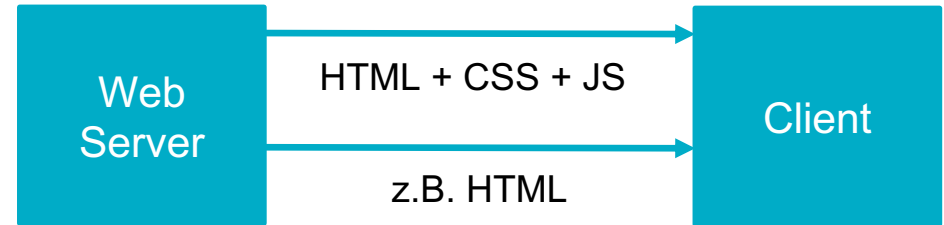
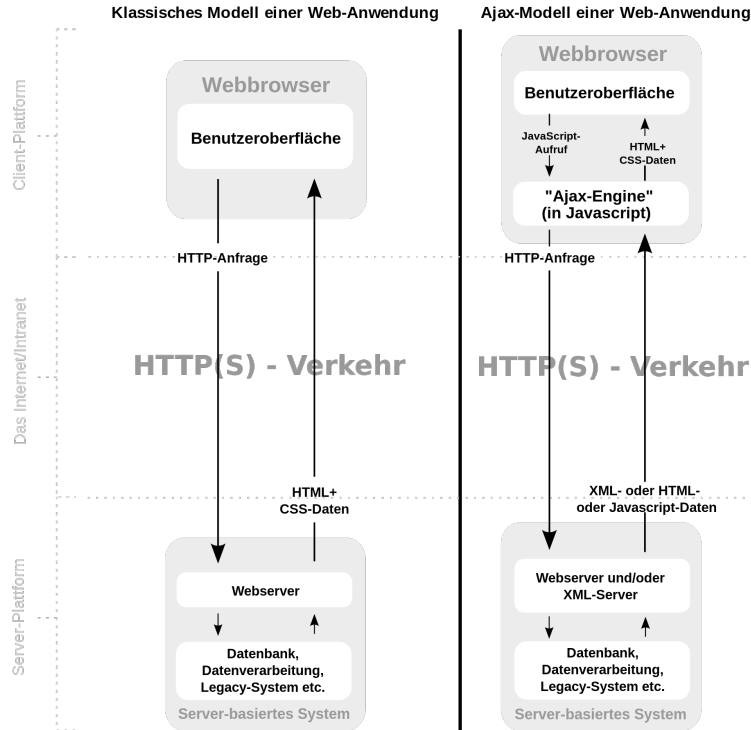
# SPA – Evolution

- **1990 – HTML**
- 1993 – CGI
- **1995 – JavaScript, PHP**
- 1996 – ASP
- **1998 – CSS2**
- 1999 – JSP
- 2002 – ASP.NET
- 2004 – Ruby on Rails
- **2006 – jQuery**
- 2008 – Draft HTML5
- **2009 – AngularJS**
- 2010 – KnockoutJS / Backbone.JS
- **2016 – Angular / React / Vue.js**

# SPA Evolution – Bis 2005



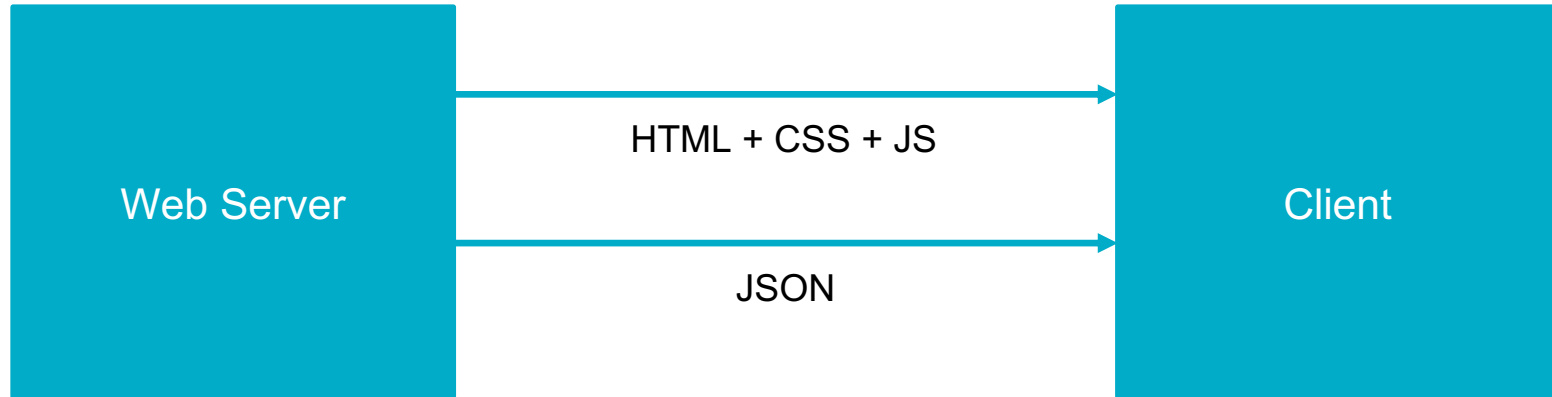
# SPA Evolution – Ab 2005



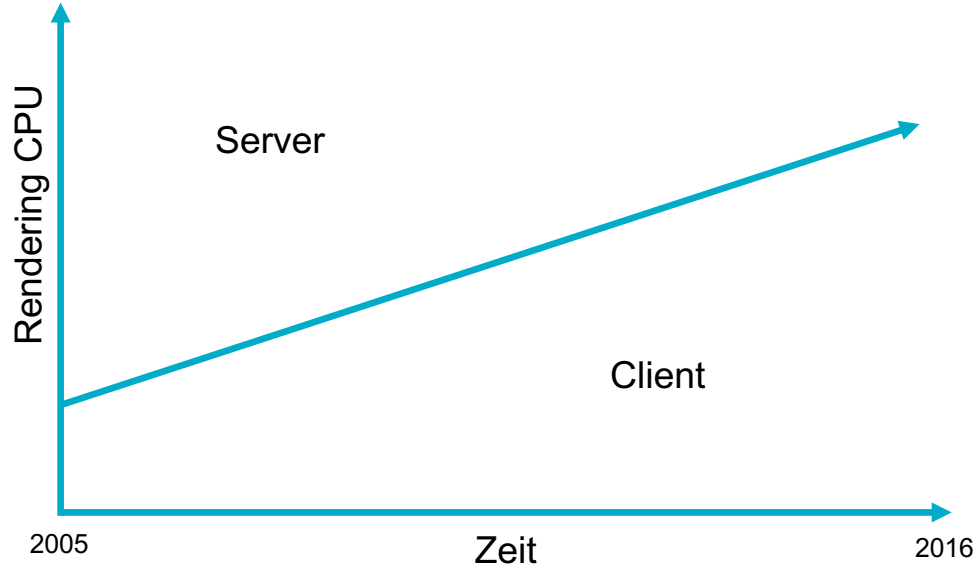
## **Asynchronous JavaScript and XML (AJAX)**

ermöglicht es, HTTP-Anfragen durchzuführen, während eine HTML-Seite angezeigt wird, und die Seite zu verändern, ohne sie komplett neu zu laden.

# SPA Evolution – Ab 2010



# SPA Evolution – Renderzeit Client/Server





# SPA Evolution

- **Browser als Betriebssystem**
  - Jedes Device hat einen Browser
  - Meta über iOS, Android, Windows, OSX
- Immer mehr **browserbasierte Anwendungen**
  - Start 2004 mit Gmail
  - Office 365, Google Docs etc.
- Ausbau **Hardware Zugriff** via Browser

# Definition SPA

**“Web App that fits on a single web page providing a fluid UX by loading all necessary code with a single page load.”**



# Charakteristiken SPA

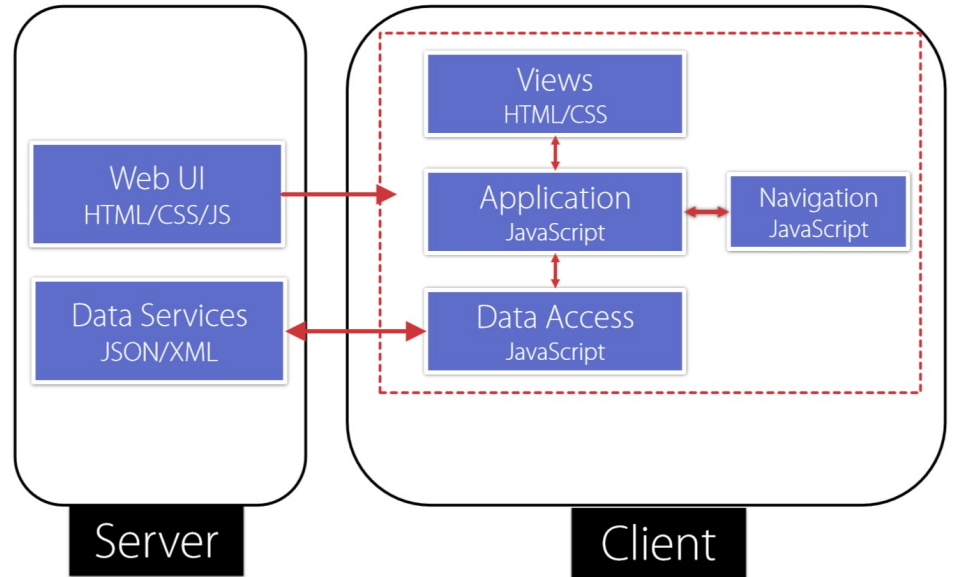
- Rich Client Application im Browser
- Reines HTML und JavaScript (kein Plugin)
- Kein erneuter Page Reload
- Navigation via Browser-Buttons funktioniert
- Links können als Bookmarks gespeichert werden
- Offline-Fähigkeit
- Interaktion mit Server via Restful Web Services

# Vorteile einer SPA

- Roundtrips reduzieren
- Reichweite der Anwendung via Browser erhöhen
- Verbesserung der User Experience

# SPA Komponenten

- Template-Rendering (View)
- Logik (z.B. Validierungen)
- (Remote-) Daten Zugriff
- Routing (Navigation)



# Fragen zu SPA

1. Was hat sich im Laufe der Zeit in Bezug auf die Architektur von Webanwendungen verändert?
2. Was war die Erfindung, die SPA's überhaupt möglich machte?
3. Wohin streben die Entwicklungen des Browsers?
4. Was sind die Vorteile einer SPA?
5. Was sind die Bestandteile einer SPA?

# Input

# Angular



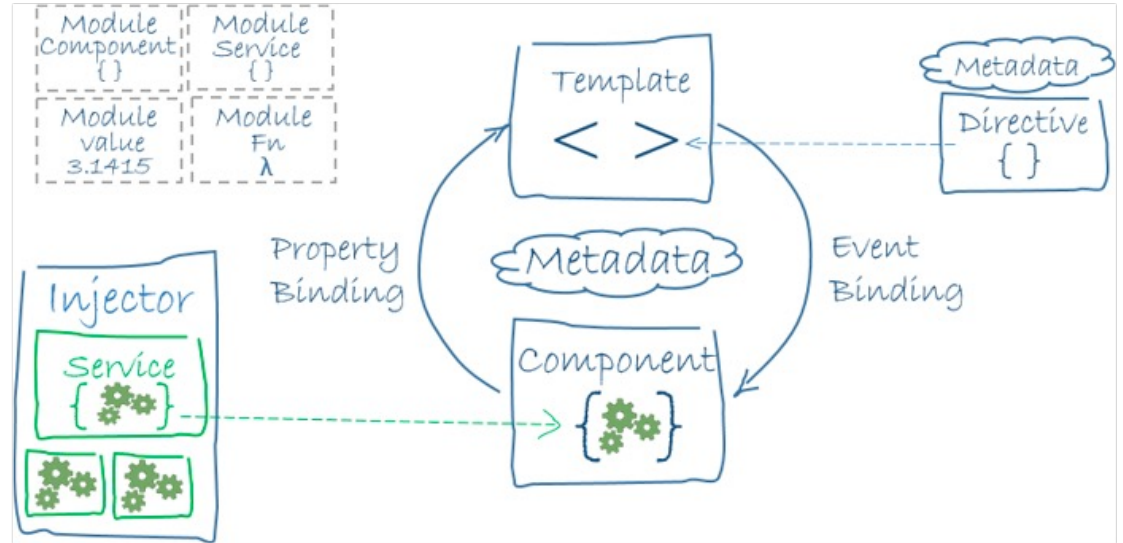
# Angular

- Ein SPA Framework für CRUD Anwendungen
- Basierend auf TypeScript
- Wenig Boilerplate Code
- Dependency Injection Mechanismus
- Performantes 2-Way Binding
- Modularisierung
- Ready-to-use Komponenten
- Testing Utils (Unit & E2E)
- Flexibel mit TypeScript oder ES8 nutzbar
- Full Stack (Angular Universal)



# Angular Architektur

- Module
- Komponenten
- Templates
- Metadaten
- Data Bindings
- Direktiven
- Services
- Dependency Injection



# Angular CLI

- Angular Command Line Interface (“**ng**”)
- Installation via NPM
- Erstellen von neuen Angular Anwendungen: “**ng new**”
- Generierung von Angular Bausteinen (Components, Routes, Services & Pipes): “**ng generate**”
- Live-Reloading während der Entwicklung: “**ng serve**”
- Bauen der Anwendung: “**ng build**”
- Testen & Linting der Anwendung: “**ng test**” / “**ng e2e**”

# Angular CLI

```
> npm install -g @angular/cli  
  
> ng new my-dream-app  
  
> cd my-dream-app  
  
> ng serve
```

**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/angular-seed  
→ cd ./angular-seed && npm install  
→ npm run start
```

# Module

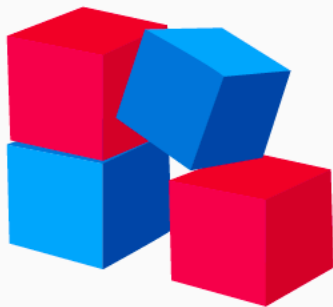
- Angular-spezifisches Konstrukt zur Organisation der Anwendung.
- Ein Modul sollte eine in sich geschlossene Einheit bilden mit einem eindeutigen Verwendungszweck.
- Module haben eine interne und eine externe Sicht
  - Interne Sicht: Alle Informationen die benötigt werden, um eigene Komponenten, Services, Direktiven oder Pipes zu erstellen.
  - Externe Sicht: Beinhaltet alle Komponenten, Services, Direktiven oder Pipes, die anderen Modulen zur Verfügung gestellt werden.

# Module

```
@NgModule({
  imports: [
    BrowserModule,
    ReactiveFormsModule
  ],
  declarations: [
    AppComponent,
    TopBarComponent,
    ProductListComponent
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

# Component

- Grundbaustein einer Angular App
- Jede Angular App ist als Hierarchie von Komponenten aufgebaut.





# Component

- Jede Komponente besteht aus der Businesslogik, dem Template und dem Styling.
- Eine Komponente kann via Lifecycle Hooks gesteuert werden und wird von Angular zur Laufzeit verwaltet (erstellt, aktualisiert und entfernt).
- Angular Komponenten können exportiert und in anderen Modulen oder Anwendungen eingesetzt werden.

# Component

*app.component.ts*

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

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

*app.component.html*

```
<app-top-bar></app-top-bar>

<div class="container">
  <p>Hallo</p>
</div>
```

*app.component.css*

```
.container {
  font-family: Lato;
}
```

# Component: Data Binding

- Verbindet das Template (HTML) mit der Businesslogik (TS)

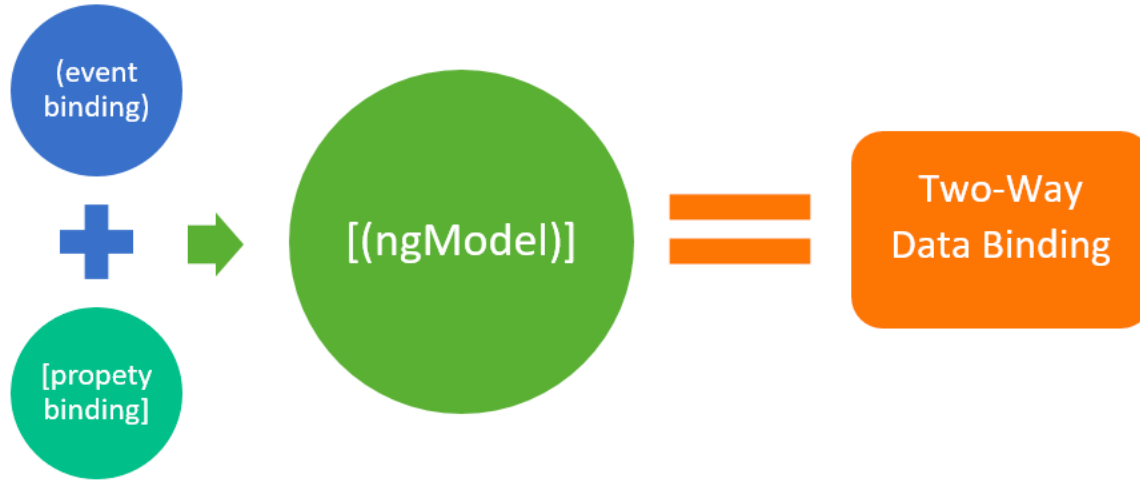
```
<!-- Interpolation -->
<div>{{product.name}}</div>

<!-- Property Binding -->
<product-detail [product]="product"></product-detail>

<!-- Event Binding -->
<div (click)="selectProduct(product)"></div>
```

- {{product.name}} zeigt den product.name Wert der Komponente im "li"-Element an (Interpolation).
- Mit [product] wird das Objekt "product" der Komponente an das "product" Property der Child Komponente "product-detail" übergeben (Property Binding).
- Der (click) wird bei einem Klick auf das "div" die Methode "selectProduct" der Komponente aufgerufen (Event Binding).

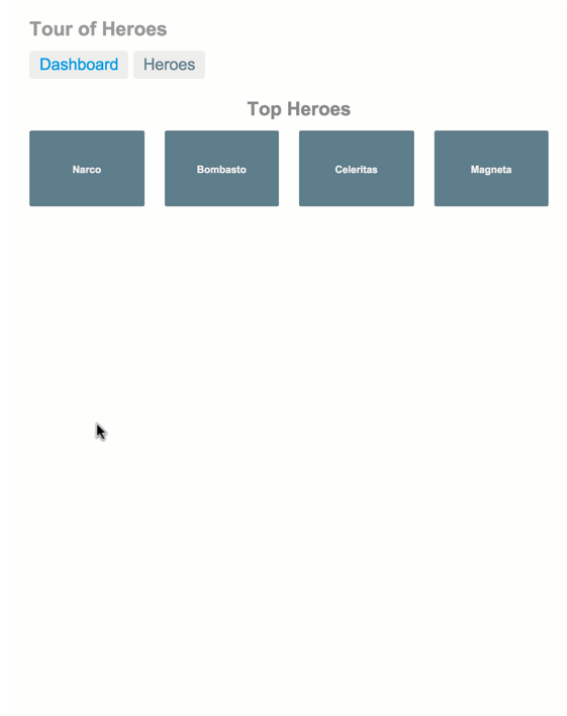
# Component: Two-Way Data Binding



```
<input [(ngModel)]="product.name">
```

# Übung: Angular Tour of Heroes

- Offizielle Übung von angular.io zum Kennenlernen der Core Features.



# Übung: Angular Tour of Heroes Part 0+1

- The Application Shell: <https://angular.io/tutorial/toh-pt0>
- The Hero Editor: <https://angular.io/tutorial/toh-pt1>

# Input

## Angular Directives



# Directive

- Direktiven sind Befehle, die das Aussehen des Templates anpassen.
- 3 verschiedene Arten:
  - Attribute Direktiven
  - Strukturelle Direktiven
  - Komponenten



# Directive

- Direktiven sind Befehle, die das Aussehen des Templates anpassen.
- 3 verschiedene Arten:
  - **Attribute Direktiven:** Änderung der Attribute eines DOM Elements
  - Strukturelle Direktiven
  - Komponenten

# Directive

- Direktiven sind Befehle, die das Aussehen des Templates anpassen.
- 3 verschiedene Arten:
  - Attribute Direktiven
  - **Strukturelle Direktiven:** Struktur-Anpassung des DOM Trees
  - Komponenten

# Directive

- Direktiven sind Befehle, die das Aussehen des Templates anpassen.
- 3 verschiedene Arten:
  - Attribute Direktiven
  - Strukturelle Direktiven
  - **Komponenten**: Grundbaustein einer Angular App

# Directive

- Direktiven sind Befehle, die das Aussehen des Templates anpassen.
- 3 verschiedene Arten:
  - **Attribute Direktiven:** Änderung der Attribute eines DOM Elements
  - **Strukturelle Direktiven:** Struktur-Anpassung des DOM Trees
  - **Komponenten:** Grundbaustein einer Angular App

# Directive: Standarddirektiven

- Listen mit \*ngFor (Strukturelle Direktive)

```
<li *ngFor="let item of [1, 2, 3]">  
  {{ item }}  
</li>
```

- Fallunterscheidungen mit \*ngIf (Strukturelle Direktive)

```
<div *ngIf="true">True</div>  
<div *ngIf="false">False</div>
```

- Styles verändern mit ngStyle (Attribute Direktive)

```
<div [ngStyle]="{ 'color': 'red' }"> Beispiel für ngStyle </div>
```

# Directive: Stern-Syntax

- Abgekürzte Schreibweise für strukturelle Direktiven.

Kurzform:

```
<div *ngIf="person" class="name">{{person.name}}</div>
```

Langform:

```
<ng-template [ngIf]="person">  
  <div class="name">{{person.name}}</div>  
</ng-template>
```

# Directive: Eigene Attribute Direktive

```
import { Directive, ElementRef } from '@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {

  constructor(el: ElementRef) {
    el.nativeElement.style.backgroundColor = 'yellow';
  }

}
```

```
<div appHighlight> Yea Yellow! </div>
```

# Übung: Angular Tour of Heroes Part 2+3

- Displaying a List: <https://angular.io/tutorial/toh-pt2>
- Master/Detail Components: <https://angular.io/tutorial/toh-pt3>



# Input

# Angular Services & Dependency

# Injection



# Service

- Ein Service in Angular ist eine TypeScript Klasse mit einer bestimmten Aufgabe, z.B:
  - Bereitstellung von Daten
  - Kommunikation zwischen Komponenten
  - Wiederverwendbarkeit von Funktionen

# Service

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

@Injectable()
export class MyService {

  constructor() { }

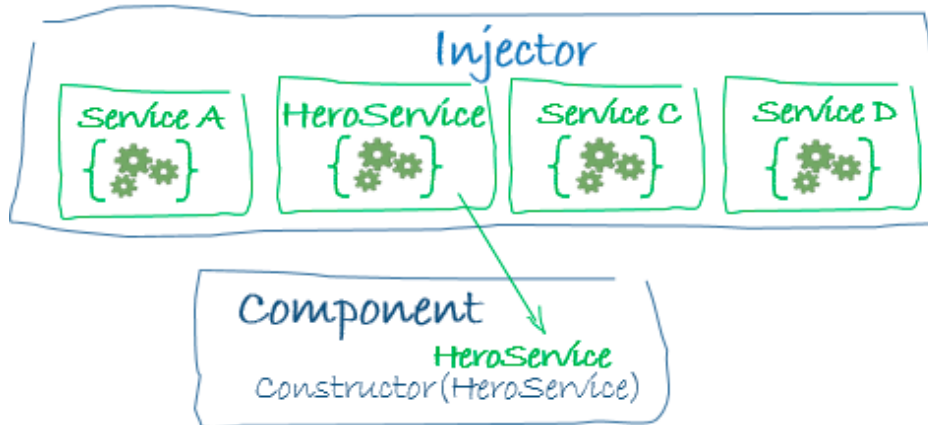
}
```

Nutzung des Services in einer Komponente via Dependency Injection:

```
constructor(private myService: MyService) { }
```

# Dependency Injection

- Zentrale Kontrolle der Instanziierung von Direktiven und Services mittels Injector.
- Der Injector kümmert sich um die richtige Reihenfolge der Dependencies.



# Dependency Injection: Vorteile

- Der Code ist modularer und wiederverwendbar
- Der Code ist meistens wartbarer
- Der Code einfacher zu testen
- Das API ist simpler und abstrakter

# Dependency Injection

```
import { Injectable } from '@angular/core';  
  
@Injectable()  
export class HeroService {  
  
    constructor() { }  
  
}
```

Nutzung des Services in einer Komponente:

```
constructor(private heroService: HeroService) { }
```

# Dependency Injection: Singleton Service

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

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

  constructor() { }

}
```

Nutzung des Services in einer Komponente:

```
constructor(private heroService: HeroService) { }
```

# Übung: Angular Tour of Heroes Part 4

1. Services: <https://angular.io/tutorial/toh-pt4>



# Input

## Angular Pipes



# Pipes

- Zur Formatierung von Daten vor der Darstellung.
- Transformation von Daten, ohne den Ausgangsdatensatz zu verändern.
- Angular stellt Built-in Pipes zur Verfügung
- Eigene Pipes können via Custom Pipes erstellt werden
- Beispiele:
  - Ausgabe eines Datums
  - Formattierung von Geldbeträgen

# Demo

≡

ANGULAR

FEATURES



DOCS

RESOURCES

EVENTS

BLOG

pipes



Introduction

Getting Started >

Understanding Angular >

Developer Guides >

Best Practices >

Angular Tools >

Tutorials >

Release Information >

Reference ✓

Conceptual Reference >

CLI Command Reference >

API Reference

Error Reference >

Example applications

Angular Glossary

Angular Style and Usage >

## Pipes

AsyncPipe	Unwraps a value from an asynchronous primitive.
CurrencyPipe	Transforms a number to a currency string, formatted according to locale rules that determine group sizing and separator, decimal-point character, and other locale-specific configurations.
DatePipe	Formats a date value according to locale rules.
DecimalPipe	Transforms a number into a string, formatted according to locale rules that determine group sizing and separator, decimal-point character, and other locale-specific configurations.
I18nPluralPipe	Maps a value to a string that pluralizes the value according to locale rules.
I18nSelectPipe	Generic selector that displays the string that matches the current value.
JsonPipe	Converts a value into its JSON-format representation. Useful for debugging.
KeyValuePipe	Transforms Object or Map into an array of key value pairs.

@angular/common

Entry points

Primary

Secondary

Primary entry point exports

NgModules

Classes

Functions

Structures

Directives

Pipes

Types

# Custom Pipes

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({name: 'exponentialStrength'})
export class ExponentialStrengthPipe implements PipeTransform {
  transform(value: number, exponent?: number): number {
    return Math.pow(value, isNaN(exponent) ? 1 : exponent);
  }
}
```

```
<p>Super Hero Power: {{power | exponentialStrength: factor}}</p>
```

## Power Boost Calculator

Normal power:

Boost factor:

Super Hero Power: 5

# Übung: Pipe

Erstelle in deiner TOH App eine Pipe zum Kürzen von Hero Namen:

- Die Anzahl Zeichen sollen konfigurierbar sein
- Es soll auch möglich sein, ein Suffix wie «...» zu definieren, falls der Name gekürzt wird.
- Nutze keine vorhandenen Pipes

Beispiel-Text: «Superman»

Beispiel-Output (Textlänge 3, Suffix '...'): «Sup...»

# Input Routing



# Routing

- **Routing:** Navigation der Website
- **Server-Side Routing:**
  - Via HTTP GET-Request wird eine neue Page vom Server geladen. Die URL mappt direkt auf Methoden/Objekte des Servers.
- **Client-Side Routing:**
  - Das Routing läuft im JavaScript der Website ab, ohne eine neue Page vom Server zu laden.
  - Die URL wird wie beim Server-Side Routing angepasst. Dies resultiert in einem State Change, welcher sich auf die View auswirkt.

# Angular Router

- Ermöglicht **Seitennavigation**, ohne die Seite neu zu laden.
- Benutzerverhalten wie bei klassischer Website:
  - Anpassung der URL
  - Verwendung der Browser History
- Durch Konfiguration kann festgelegt werden, welche Komponente bei welcher Route dargestellt werden soll.



# Angular Router: package.json

```
{  
  "name": "angular-seed",  
  "version": "0.0.0",  
  "private": true,  
  "dependencies": {  
    "rxjs": "6.5.2",  
    "@angular/core": "8.2.0",  
    "@angular/common": "8.2.0",  
    "@angular/router": "8.2.0»  
    ...  
  }  
}
```

# Angular Router: Router Configuration

```
import ...

const appRoutes: Routes = [
  { path: 'test1', component: Test1Component },
  { path: 'test2', component: Test2Component },
  { path: '**', component: PageNotFoundComponent }
];

@NgModule({
  imports: [
    RouterModule.forRoot(appRoutes, { enableTracing: true } // <-- debugging purposes),
    BrowserModule, FormsModule
  ],
  declarations: [ AppComponent, PageNotFoundComponent, Test1Component, Test2Component ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

# Angular Router: index.html

```
<head>  
  <base href="/">  
</head>  
  
<body>  
  <app>Loading...</app>  
</body>
```

# Angular Router: app.component.html

```
<h1>Angular Router</h1>

<nav>
  <a routerLink="test1">Test1</a>
  <a routerLink="test2">Test2</a>
</nav>

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

# Übung: Angular Tour of Heroes Part 5

1. Routing: <https://angular.io/tutorial/toh-pt5>

# Input

# Http



# Asynchronität in Web Anwendungen

- **Callback** - Funktion, die einer anderen Funktion übergeben und von dieser aus aufgerufen wird.
- **Event** - Interaktion des Nutzers (z.B. click), Kommunikation zwischen Komponenten (EventEmitter)
- **Promise** - kapselt Asynchrone Code-Blöcke, kann erfolgreich sein oder fehlschlagen.
- **Observable** - Stream, wichtiger Bestandteil der reaktiven Programmierung.

# Observables

- Observables sind Bestandteil der “Reactive Extensions for JavaScript”, kurz **RxJS**.
- 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 und damit die später ankommenden Daten. Ein Observable kann gefiltert, gruppiert oder transformiert werden.



# Observable im Angular Service

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

@Injectable()
export class PizzaService {

    constructor(private http: HttpClient) {
    }

    getPizza(): Observable<any> {
        return this.http.get('assets/pizza.json');
    }
}
```

# Observable Subscription in der Komponente

```
import { Component } from '@angular/core';
import { products } from '../products';
import { PizzaService } from '../top-bar/top-bar.component';

@Component({
  selector: 'app-product-list',
  templateUrl: './product-list.component.html',
  styleUrls: ['./product-list.component.css']
})
export class ProductListComponent {
  products = products;
  pizzas: any[];

  constructor(private pizzaService: PizzaService) {
  }

  loadData() {
    this.pizzaService
      .getPizza()
      .subscribe((pizzas: Array<Object>) => this.pizzas = pizzas);
  }
}
```

# HttpClient

- Bestandteil des Moduls “**angular/common/http**”
- Abstraktion von XMLHttpRequest
- **Bestandteile:**
  - Testbarkeit
  - Typisierte Request/Response Objekte
  - Request/Response Interception
  - Observable API
  - Error Handling

# HttpClient

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

@Injectable()
export class PizzaService {

  constructor(private http: HttpClient) {
  }

  getPizza(): Observable<any> {
    return this.http.get('assets/pizza.json');
  }
}
```

# HttpClient

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

@Injectable()
export class PizzaService {

    constructor(private http: HttpClient) {
    }

    getPizza(): Observable<any> {
        return this.http.get('assets/pizza.json');
    }
}
```

# Übung: Angular Tour of Heroes Part 6

1. HTTP: <https://angular.io/tutorial/toh-pt6>

# 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/>