

# Angular Komplett

## Saban Ünlü

# Zwei Worte zu mir

# Saban Ünlü

- Software Architekt und Programmierer
- Berater und Dozent seit 2000
- Autor
- Influencer
- Gründer von netTrek

# jQuery

# JavaScript Bibliothek

# Überblick

- 2006 Entwickelt von John Resig
- Web 2.0 Internet bekommt eine neue Rolle
  - Benutzer als Akteur
- Ajax (**A**synchronous **J**ava**S**cript **a**nd **X**ML)
  - Vorreiter der SPA (**S**ingle **P**age **A**pplication)
- Auf Basis von XML
- Heute weiterentwickelt von der jQuery Foundation

# Vorteile

- Browser Übergreifend
- Filtern und Finden
- DOM Manipulation
- Ereignisse
- XHR & XML

# Initialisieren

- jQuery Bereitstellen
  - Download, CDN, NPM uvm.
- Einbinden im Script Block
  - `<script src="jquery.js"></script>`

# Syntax

- `$(selector).action()`
- `$` === `jQuery`, globale Referenz
- selector: Query, HTML-String oder HTML-Element
- Rückgabe entspricht einer `jQuery-Instanz` oder Liste von `jQuery-Instanzen`, auf der Aktionen ausgeführt werden.
- `action()` z. B. für Dom Manipulationen u.v.m.
  - Gib jQuery Referenz zurück

# Syntax

- `$.action()`
- für globale jQuery `action()`
- z. B. zum Laden via GET-Methode

# Konflikte vermeiden

- \$ kann bereits in Verwendung sein.
- var \$jq = **jQuery.noConflict();**
- noConflict gibt eine isoliert, Konfliktbefreite Version der jQuery Referenz zurück.
  - Unmittelbar nach dem Laden

# Initialisieren - onload

- Laden der Website abwarten
  - Ohne: `window.onload = () => {};`
  - Mit: `$( document ).ready( ()=> {} );`
  - die `ready` Aktion erwartet ein **Callback Handler**
    - Aufgerufen wenn HTML vollständig initialisiert.

# Selectoren

- Analog `querySelector`
  - Elemente, CSS-Klassen, Attribute u.v.m.
  - HTML-Element
    - `$( 'section' )`
  - Kinder innerhalb der Unterstrukturen
    - `$( 'section h1' )`

# Selectoren

- Element mit ID
  - `$ ('#id' )`
  - `$ ('section#id ')`
- Element mit CSS-Klasse
  - `$ ('.css' )`
  - `$ ('section.css')`

# Selectoren

- Element mit Attribut
  - `$('input[type]')`
- Element mit Attribut-Wert
  - `$('[type="text"]')`
  - `$('input[type="text"]')`

# Selectoren

- Element mit Attribut-Wert
- `$('[type!="text"]')` // nicht
- `$('[type~="text"]')` // enthält Ausdruck als Wort
- `$('[type^="text"]')` // beginnt mit
- `$('[type$="text"]')` // endet mit

# Selectoren

- Selectoren-Liste
- `$( '[type], section' )`
- Selector mit Pseudo-Klasse
- `$( 'selector:first' )`
  - `:visible, :odd, :gt(2), :checked, :disabled u.v.m.`
- `$( 'selector:first' ).find ('selector')`

# Manipulationen

- Aktionen zum manipulieren von DOM Elemente agieren als Getter, wenn kein Parameter definiert wird
- `$('.section').attr ()`
- `$('.a').attr ('href', 'http://netTrek.de')`
- `$('.img').attr ( {'src': 'http://netTrek.de', 'alt': ,mein Bild'})`

# Manipulationen

- Text-Container bearbeiten
- `$('.button').text ()`
- HTML-Inhalt bearbeiten (innerHTML)
- `$('.p').html ('<strong>hello</strong>')`
- Formularwerte
- `$('.input').val ( value? )`

# Manipulationen

- Position ermitteln
- `$( 'button' ).position () // Positions-Objekt { left, top}`
- Breite
- `$( 'p' ).width( value? )`
- Höhe
- `$( 'p ' ).height ( value? )`

# Manipulationen

- Elemente erzeugen
- `$('<p>hello world</p>')`
- Elemente erzeugen und in den Dom
- `$('<p>hello world</p>').appendTo( $(,body') );`
- Elemente umhängen
- `$('li').first().appendTo( $('ul') );`

# Manipulationen

- Style-Attribute
- `$(selector).css ('color')`
- `$(selector).css ('color', ,blue')`
- `$(selector).show()`
- `$(selector).hide ()`

# Manipulationen

- Aktioen für CSS-Klassen
- `$( 'selector' ).addClass (' blue ');`
- `$( 'selector' ).removeClass (' blue ');`
- `$( 'selector' ).toggleClass (' blue ');`
- `$( 'selector' ).hasClass (' blue ');`

# Ereignisse

- jQuery verfügt über diverse Aktionen, um Ereignisse zu registrieren
- `$('selector').click ( evt => {...} ); // dblclick`
- `$('selector').mouseover ( evt => {...} ); // mouseout`
- `$('selector'). mouseenter ( evt => {...} ); // mouseleave`
- `$('selector'). resize ( evt => {...} ); // scroll`



# Ereignisse

- `$( 'selector' ). focus ( evt => {...} ); // blur`
- `$( 'selector' ). change ( evt => {...} ); // select`
- `$( 'selector' ). keydown ( evt => {...} ); // keyup`
- `$( 'selector' ). contextmenu ( evt => {...} );`

# Helper-Funktionen

- Zeichenketten manipulieren
- `$.trim (, hallo '); // ,hallo'`
- Objekte iterieren
- `$.each ( listOrObj, callback ( indexOrKey, value);`
- Objekte erweitern
- `$.extend ( obj1, obj2 ); => newObj`

# Helper-Funktionen

- Type-Funktionen
- `$.isArray ( [] );`
- `$.isFunction ( ()=>{} );`
- `$.isNumeric ( 123 );`
- `$.type ( 123 ); // number`

# AJAX

- HTTP Request mit jQuery
- ```
$.ajax( url, {
    url: url,                      // endpoint
    type: method,                  // GET, POST, PUT....
    dataType: type,                // json
    data: data,                     // optionaler Payload für Post/Put
    success: callback
} ) => jqXHR;
```

# AJAX

- Rückgabe ermöglicht Registrierung weitere Callbacks
- `$.ajax(...).done ( ( data, state, response ) => {...} )`
- `$.ajax(...).fail ( response => {...} )`
- `$.ajax(...).always ( ( data, state, response ) => {...} )`

# Shortcut Methode

- Stellt jQuery zum schnellen Reques bereit
  - get Aufruf
    - `$.get( url, opts? ) => jqXHR`
  - Post Aufruf
    - `$.post( url, payload , opts? ) => jqXHR`

# Daten

- jQuery Objekte besitzen einen Daten-Container
- `$(selector).data('key', 'value'); // schreiben`
- `$(selector).data('key'); // lesen`

# TypeScript

# Grundlagen

- Programmiersprache basierend von ES6 (ES2015)
  - Entwickelt durch Microsoft
- Exporte in andere ECMA-Script Versionen
- Export in unterschiedliche Modul Handlings
- Typsicherheit
- Nutzung Experimenteller Annotationen

# Variablen

- Definition
  - let
  - const
- Typen
  - Native-Typen
  - Datentypen

# Klassen

- Klassen als Schablone eines JS Objektes
- **constructor**
- Eigenschaften und Methoden
- Instanziieren
- Setter und Getter
- Parameterübergabe

# Vererbung

- Klassen können von anderen Klassen erben
  - **extends**
- Gültigkeitsbereiche
  - **private, public & protected**
- Überschreiben
  - **super**

# Interfaces

- Interfaces sind die Schablonen einer Klasse
  - Interfaces können erben - **extends**
- Implementiert wird ein Interface über
  - **implements**

# Abstrakte Klasse

- Implementieren Basis Funktionen und Eigenschaften
- Dient als Vorlage für ein Derivat (Vorlage)
- Kann nicht instanziert werden

# Syntax Magie

- Syntax magic (ES6/TS)
  - private, public definition in constructor
  - Concat Array
  - Object Assign
  - Destructuring

# Angular

## Einleitung

# Einleitung

- Was ist Angular?
- Angular: Die Highlights
- Was ist neu?
- Unterschiede zu AngularJS 1x
- Technologien
- Polyfills und Vendors
- Angular-Module

# Webapplikationsframework

- JavaScript-Framework (<https://angular.io/>)
  - Für moderne Desktop- und Mobile-Webanwendungen
  - Derivate für App-Entwicklung ( NativeScript || ionic )
- Unterstützt mit Konventionen und Richtlinien
  - Vereinfacht Teamwork

# Webapplikationsframework

- Unterstützt viele Funktionen wie z. B.:
  - Client-Server-Kommunikation
  - Datenbindung
  - Animationen
  - Routing
  - Single-Page-Application

# Was ist Angular?

- Framework für Single Page Application
- Komponentenbasierte Anwendungen
  - inspiriert von Web-Components
- Modulare Arbeitsweise
- Trennung von Logik und View

# Angular: Die Highlights

- Vorlagen
- Bindungen
- Services
- Dependency Injection
- Routing
- Formulare

# Was ist neu?

- Angular ist kein klassisches Update
- Komplett neu geschrieben
- Fokus
  - Performance (3-5 mal schneller als Angular 1)
  - Komponenten
  - Modulare Arbeitsweise

# Unterschiede zu AngularJS 1x

- Komponenten statt Kontroller
  - Wie Element-Direktiven in Angular 1
- Verzicht auf Scope
  - Bindung erfolgt auf Komponenten-Instanz

# Unterschiede zu AngularJS 1x

- Entwicklung mit ES5, Dart oder TypeScript
- Angular wurde mit TypeScript entwickelt
  - ES2015: Klassen, Interfaces und Vererbung
  - ES2015: Templates
  - Typisiert
  - Annotations

# Unterschiede zu AngularJS 1x

- Angular ist mit TypeScript ES2015 konform
- TypeScript wird für aktuelle Browser in ES5 ausgegeben
- ES2015 Polyfills für “weniger moderne” Browser
- System zum Laden und Verwalten von Modulen z.B.:
  - webpack

# Technologien

# Technologien im Überblick



# Node.js

- JavaScript-Laufzeitumgebung
- Verfügbar für unterschiedliche Betriebssysteme
- Benötigt:
  - Testen
  - Veröffentlichen

# TypeScript

- Auf ES2015 basierende Programmiersprache
  - Klassen, Vererbung, Typisierung, Interface, Enum uvm.
- Exportiert auf ES5
- Angular wurde mit TypeScript entwickelt

# git

- Versionierungssystem für Software
- GitHub – Filehoster
- Ermöglicht, unterschiedliche Zustände einer Software zu verwalten
- Optimiert Teamwork

# webpack

- Bündelt statische Inhalte in Pakete
- Im Angular-Kontext
  - ES-Module, Styles, Vorlagen
    - JavaScript-Pakete
- Vereinfachte Veröffentlichung
- Optimierte Ladeprozesse

# SASS

- Erweiterungssprache für CSS
  - Präprozessor für CSS
- Unterstützt
  - Variablen, Funktionen, Erweiterung, Imports uvm.
- Sehr steile Lernkurve

# Jasmine

- Entwicklungs-Framework zum Testen von JavaScript-Code
  - Unabhängig von weiteren Frameworks
  - Benötigt kein DOM
- Ermöglicht die Definition von verhaltensorientierten Tests
  - Erwartung wird definiert und geprüft
    - `expect(a).toBe(true);`

# Karma

- Framework zum Steuern von JavaScript-Tests
  - Bereitgestellt vom Angular-Team
  - Unterstützt: Jasmine, Mocha und QUnit
- Ermöglicht das Testen auf Geräten
- Sehr gute Integration in Continuous Integration z.B. mit Jenkins

# Protractor

- Framework für End-to-End-Tests
  - Entwickelt von Google für Angular
  - Tests im echten Browser
  - Simuliert einen Benutzer
  - Benutzerereignisse z. B. Klicks oder Eingaben
  - Wartet auf asynchrone Ereignisse

# Polyfills

- JavaScript-Files
- Überprüft die Existenz bestimmter Funktionen in Browsern
- Falls nicht vorhanden, wird die Funktion erweitert
  - Workaround für ältere Browser

# core-js

- Polyfill für ES6 (ES2015) Funktionen
- Häufig benötigt von weniger modernen Browsern
- Insbesondere der IE benötigt hier Hilfe
- Für die Nutzung von Dekoratoren werden im JIT-Kontext auch ES7/reflect benötigt

# web-animations

- Angular-Animationen basieren auf der 'Web Animations API'
- Die API wird bislang nur von Chrome, Firefox und Opera gut unterstützt
- Alle anderen benötigen das web-animations-Polyfill

# Zone.js

- Framework ermöglicht die Definition eines Ausführungskontexts für JavaScript
  - Vergleichbar Domains in Node.js
- Wird in Angular als Abhängigkeit genutzt
- Überwacht und steuert die Ausführung
  - Hilft beim Debugging

# ReactiveX

- Framework, um Ereignisse und asynchrone Prozesse zu überwachen
- Wird für unterschiedliche Programmiersprachen angeboten
- RxJS ist die JavaScript-Variante
- In Angular als Abhängigkeit genutzt, unter anderem für **HTTP** und **EventEmitter**

# Angular Module

# @angular/core

- Modul, das jede Angular-Anwendung benötigt
- In diesem Modul werden Kernfeatures bereitgestellt
  - Komponenten & Direktiven
    - Lebenszyklus
  - Pipes
  - Dependency Injection

# @angular/common

- Modul, das fast jede Angular-Anwendung benötigt
  - Import von `BrowserModule` oder `CommonModule`
- In diesem Modul werden allgemeine Features bereitgestellt
  - Direktiven z. B. `ngIf`
  - Pipes z. B. `date`



# @angular/compiler

- Modul, das JIT-Angular-Anwendungen benötigt
- Dieses Modul stellt den Angular-Compiler bereit
- Der Compiler erzeugt zur Laufzeit im Browser ausführbaren Code und verschmilzt Logik- und Darstellungsschicht
- Wird von platform-browser-dynamic angestoßen

# @angular/platform-browser & dynamic

- Modul, das jede Angular-Anwendung benötigt
- Steuert Browser und DOM-relevante Bestandteile zum Rendern von Elementen und Verwalten von Ereignissen
- Abhängig von der Veröffentlichungsform JIT oder AOT wird
  - platform-browser (AOT) oder
  - platform-browser-dynamic (JIT) verwendet

# @angular/platform-webworker & dynamic

- Alternatives Modul zu platform-browser
- Analog zu platform-browser, aber mit Unterstützung von Webworkern
- Abhängig von der Veröffentlichungsform JIT oder AOT wird
  - platform-webworker (AOT) oder
  - platform-webworker-dynamic (JIT) verwendet

# @angular/platform-server

- Modul, das Universal-Angular-Anwendungen benötigt
- Auf Node.js-Ebene wird die Anwendung auf dem Server ausgeführt und liefert HTML-Seiten aus
  - Kleinere Ladezeiten
  - Kürzere Startup-Zeit
  - Perfekt für Bots

# @angular/http

- Modul, das Angular-Anwendungen benötigt, die Client-Server-Kommunikation betreiben
  - Import über [HttpModule](#)
- Stellt Services bereit, um XHR-Request zu unterstützen
  - Entwicklung
  - Testen

# @angular/forms

- Modul, das Angular-Anwendungen benötigt, die Formulare anbieten
  - Import über **FormsModule** oder **ReactiveFormsModule**
- Stellt Features für die Formularnutzung bereit
  - Vorlagen-getrieben
  - Reaktiv

# @angular/router

- Modul, das Angular-Anwendungen benötigen, die als Single-Page-Application arbeiten
  - Import über **RouterModule**
- Ermöglicht die Konfiguration und Steuerung von Routen
  - Welcher **Pfad**
  - soll welche **Komponente** darstellen?

# @angular/animate

- Modul, das Angular-Anwendungen benötigt, die Animationen im Angular-Kontext nutzen.
  - Import über [BrowserAnimationsModule](#)
- Das Modul stellt die Brücke zwischen der Web-Animations-API und der Angular-Welt dar.

# Projektsetup

# Erste Schritte

- Mac
  - XCODE installieren
  - node.js installieren (>= 8.9.x)
- Win
  - node.js installieren (>= 8.9.x)
  - Git installieren (inkl. Bash)

# Setup Manuell

- Node initialisieren
- Abhängigkeiten installieren
- TypeScript konfigurieren
- Webpack konfigurieren

# Seed Setup

- git clone <https://github.com/angular/quickstart.git>  
myProject
- npm install

# angular-cli

- ng new netTrek --prefix=nt --style=scss --routing=true
- ng serve
- ng g m commonUi
- cd common-ui/
- ng g c user

# Architektur

# Einleitung

- Decorator
- Module
- Komponenten
- Bootstrap
- Direktiven
- Pipes
- Datenbindung
- Dependency Injection (DI)
- Services
- Router

# Architektur

## Decorator

# Decorator

- Funktionen mit vorangestelltem @-Symbol
- Wird vor einer Deklaration verwendet
- Decorators in Angular haben gleiche Kernfunktionalitäten
  - Speichern von Metainformationen
  - Manipulation nachfolgender Deklaration

```
@HostListener('click')
onHostClick() { /* */ }
```

# Decorator

- Decorator-Typ
  - Klassen dekorieren
  - Eigenschaften dekorieren
  - Methoden dekorieren
  - Parameter dekorieren

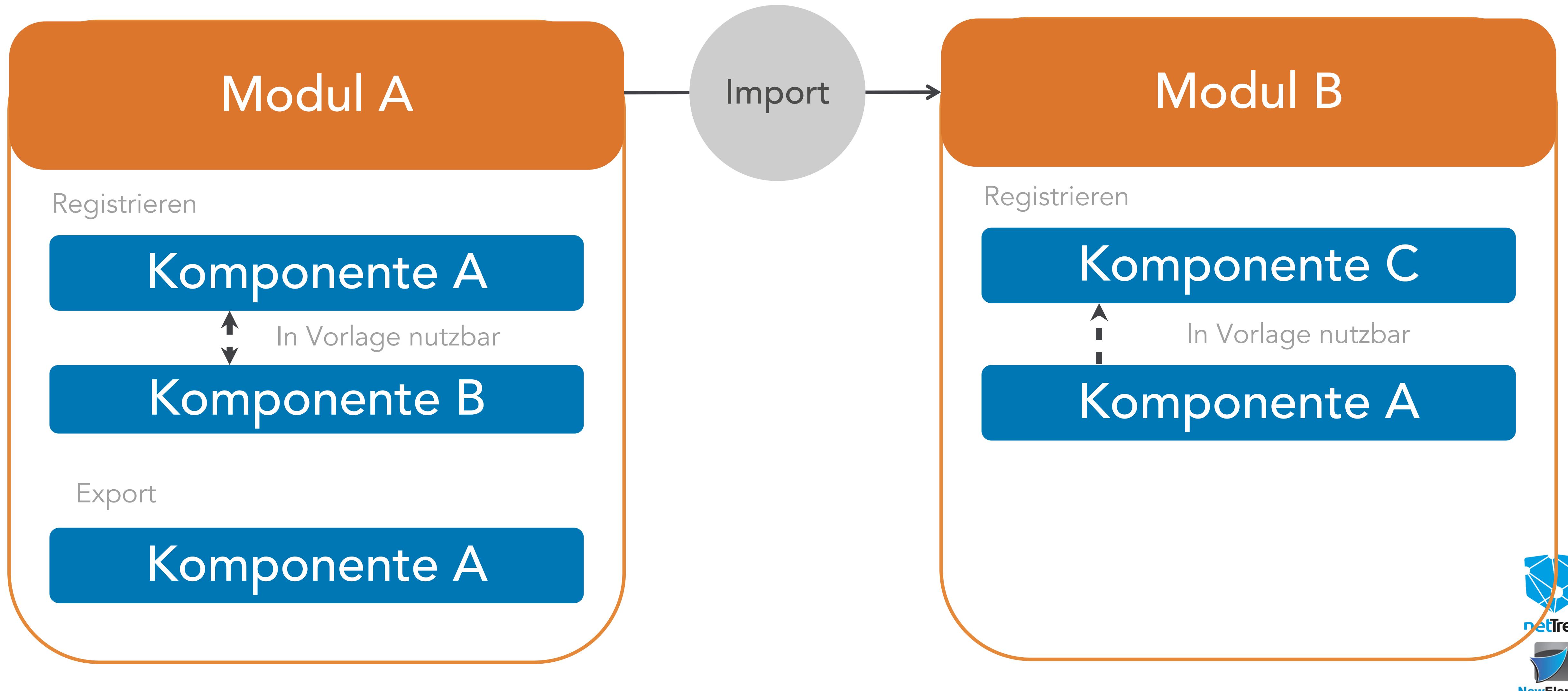
# Architektur

## Module

# Modulare Entwicklung

- Angular-Module
  - Perfekt für Teamwork
  - Wiederverwendbar
    - Export/Import
  - Container (zugänglich)
    - Komponenten, Direktiven, Pipes, Services

# Modulare Entwicklung



# Module

- Nicht vergleichbar mit JavaScript-Modulen
- Funktionen und Features in einer Black-Box bündeln
- Anwendung und eigene Module mit externen Modulen erweitern
- Compiler mitteilen, nach welchen Elementen auszuschauen ist

# Module

- Angular-eigene Module
  - BrowserModule (Ereignisse, DOM)
  - CommonModule (Direktiven, Pipes)
  - HttpClientModule (XHR)
  - FormsModule (Formulare)
  - RouterModule (Komponenten-Router)

# Module

- Module erzeugen
  - Modul-Klasse anlegen

# Module

```
class AppModule {}
```

# Module

```
@NgModule({  
  imports: [ BrowserModule ]  
})  
export class AppModule {}
```

# Module

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

# Module

- ng g m commonUi in src/app
- @NgModule
  - imports
    - definiert Module die in diesem Modul benötigt werden
  - declarations
    - benötigte Komponenten, Direktiven, Pipes

# Module

- @NgModule
  - providers
    - Bestimmt welche Service der Injector dieses Moduls für die DI bereitstellt.
  - exports
    - Exportiert Komponenten, Direktiven, Pipes dieses Moduls damit importierende Module das nutzen

# Module

- @NgModule
  - bootstrap
    - Komponenten, die beim Bootstrap dieses Moduls in den ComponentFactoryResolver abgelegt werden.  
Analog - entryComponents

# Module

- @NgModule
  - entryComponents
    - Kompiliert Komponenten bei der Definition des Moduls. Anschließend ist die Nutzung ohne Komponente-Kontext möglich, weil es als ComponentFactory und die componentFactoryResolver abgelegt wird.

# Module - Bootstrap

- in der main.ts
- platformBrowserDynamic
  - bootstrapModule
    - AppModule
    - bootstrap der Komponenten

# Architektur Komponenten

# Einleitung

- Decorator und Metadaten
- Angular Module
- Bootstrap Root-Component
- Bootstrap eine Modules
- Selector
- Vorlagen
- Styling
- Komponenten verschachteln (Shared-Modules)
- ng-content
- ViewChilds
- Lifecycle hook

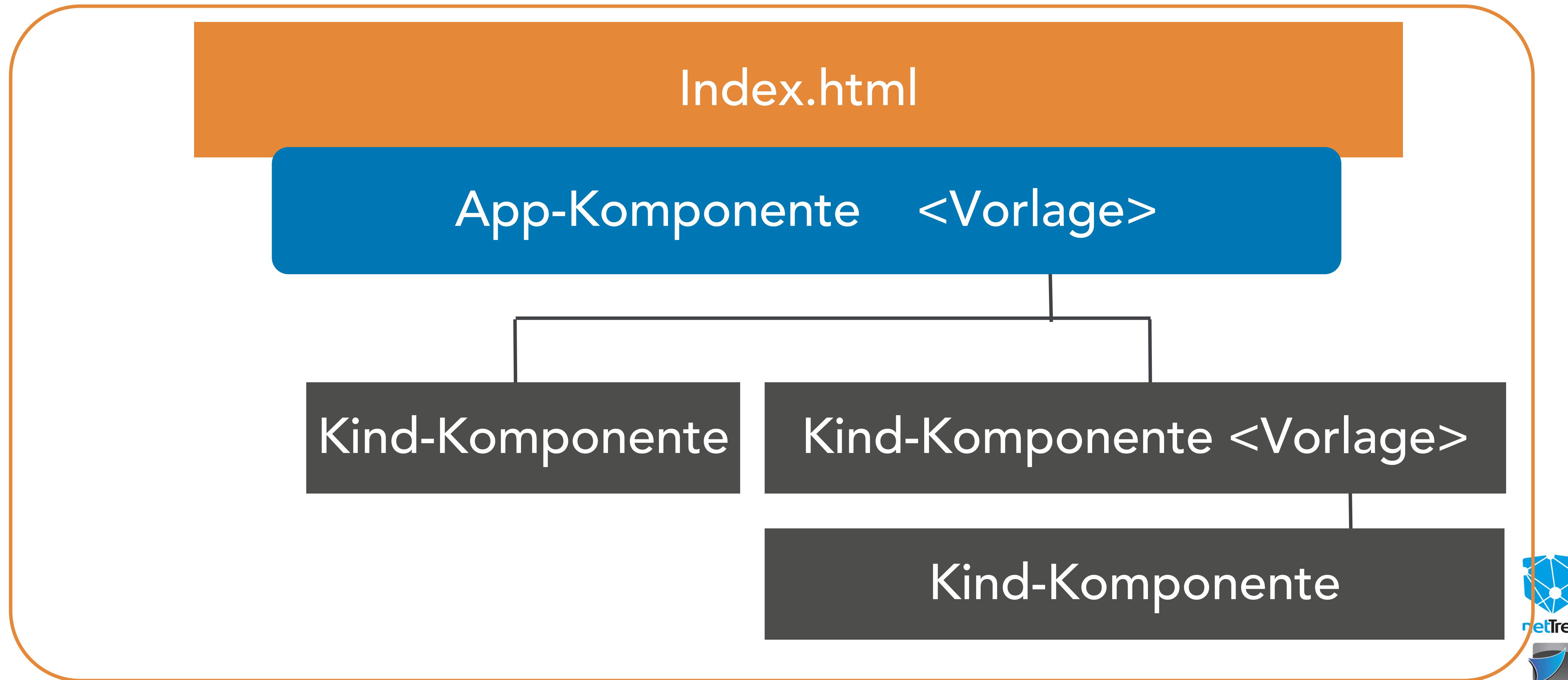
# Komponentenbasierte Entwicklung

- Komponente entspricht eigenen HTML-Knoten
  - Logik
  - Vorlage (HTML)
  - Style (optional)
- Kind-Komponente
  - Verwendung von Komponenten innerhalb einer Vorlage

# Vorlagen

- HTML-Schnipsel
  - Stellt Benutzeroberfläche einer Komponente dar
  - Definierbar als
    - Zeichenkette oder externe Dateien
    - Als Metainformation einer Komponente template oder templateUrl

# Komponentenbasierte Entwicklung



1

## Logik (TS)

```
export class UserComponent {  
  
  name = 'Saban Ünlü';  
  chgName () {  
    this.name = 'Peter Müller';  
  }  
  
}
```

2

## View (HTML)

```
<h1>{{name}}</h1>  
  
<button>(click)="chgName()">  
  Ändern  
</button>
```

3

## View (Style)

```
h1 {  
  color: darkslategray;  
}  
button {  
  background-color: yellowgreen;  
}
```

# Komponente erzeugen

- Komponenten Klasse (ts) anlegen
  - `export class ComponentName`
- Klasse mit Metainformationen versehen
  - `@Component ( { /*meta*/ } )`  
`export class ComponentName`

# Komponente erzeugen

- `@Component` – Decorator (Metainformationen)
  - `selector` – HTML-Knotenname
  - `templateUrl` oder `template` – Vorlagen der Komponente
  - `Styles` oder `styleUrls` – Liste der Style-Definitionen

# Komponente erzeugen

```
class AppComponent {  
  
    constructor () {  
        console.log ( "App Component" );  
    }  
  
}  
}
```

# Komponente erzeugen

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

selector: 'app-root',
templateUrl: './app.component.html',
styleUrls: ['./app.component.css']

export class
  name = 'app works!';
  onClick () {
    console.log ( 'clicked' );
```

# Komponente erzeugen

```
<h1 (click)="onClick()">{{name}}</h1>
```

# Komponente erzeugen

```
@NgModule({  
  imports: [ BrowserModule ] ,  
  declarations: [ AppComponent, MyComponent ]  
})  
export class AppModule {}
```

# Komponente erzeugen

```
<h1 (click)="onClick( )">{{name}}</h1>  
<my-component> </my-component>
```

# Komponent Metadaten

- ng g c user
  - selector
  - Knoten
  - Vorlage
    - templateUrl (file)
    - template (backticks)

# Komponent Metadaten

- Style
  - styleUrls (filelist)
  - styles (backtick-list)
- Spezieller Style
  - :host
  - >>>

# Komponent Metadaten

- Style
- encapsulation - Umgang mit Webkomponenten
  - ViewEncapsulation.Emulated
  - ViewEncapsulation.Native
  - ViewEncapsulation.None
  - ViewEncapsulation.ShadowDom

# Komponent Content

- Inhalte Transklusieren (transclude)
  - ng-content
  - Knoten in Vorlage
  - Attribut
  - select="nt-table-caption"

# Inhalte transkludieren

- Komponenten stellen eine View dar.
- Beschrieben wird die View in der HTML-Vorlage
- Innerhalb der Vorlage können Kinds-Komponenten mit Inhalts-Knoten versehen werden
- `<user-list>`  
`<user-item>name</user-item>`  
`</user-list>`

# Inhalte transkludieren

- Inhalts-Knoten werden transkludiert, wenn Vorlagen die **ng-content** Direktiven (Knoten) nutzen.
  - Der Knoten stellt dabei einen Platzhalter dar
  - Mittels select Attribut lässt sich definieren, für welchen Inhalt der Platzhalter greifen soll

1

## View

```
<user-list>  
  <user-header></...><br/>  
  <user-item></...><br/>  
  <user-item></...><br/>  
  <user-item></...><br/>  
</user-list>
```

2

## UserList - Template

```
<h3>user-list</h3>  
  
<ng-template><user-header></...><br/>  
  <select="user-header"><br/>  
    <user-item></...><br/>  
  </ng-content><br/>  
  <user-item></...><br/>  
<ng-template></ng-content>
```

# Zugriff auf transkludierte Inhalte

- Über Eigenschafts-Decorator
  - `@ContentChild`
    - Parameter: Komponentenklasse
    - Optionales Options-Objekt mit Read-Eigenschaft
      - `{ read:`
      - `ElementRef | ViewContainerRef | Directive | Service }`

# Zugriff auf transkludierte Inhalte

- Über Eigenschafts-Dekorator
- Zugriff erst nach Hook
  - `ngAfterContentInit`

# Zugriff auf transkludierte Inhalte

- Über Eigenschafts-Dekorator
  - @ContentChildren
    - Parameter: Komponentenklasse
    - Erzeugt
      - QueryList<Type>
        - changes -> Observable

# Vorlagen Elemente ermitteln

- Über Eigenschafts-Decorator
  - @ViewChild
    - Parameter: Komponentenklasse / Hash-ID
    - Optionales Options-Objekt mit Read-Eigenschaft
      - { read:  
ElementRef | ViewContainerRef | Directive | Service }

# Vorlagen Elemente ermitteln

- Über Eigenschafts-Dekorator
- Zugriff erst nach Hook
  - `ngAfterViewInit`

# Vorlagen Elemente ermitteln

- Über Eigenschafts-Dekorator
  - @ViewChildren
    - Parameter: Komponentenklasse
    - Erzeugt
      - QueryList<Type>
        - changes -> Observable

# Bindungen

# Bindung

- Ausdrücke interpolieren
- Eigenschaften binden
- Style-Eigenschaften binden
- CSS-Klassen binden
- Attribute binden
- Ereignisse binden
- Komponenten-Eigenschaften
- Komponenten-Ereignisse
- HostBinding
- HostListener

## Logik (TS)

```
export class UserComponent {  
  
  name = 'Saban Ünlü';  
  chgName () {  
    this.name = 'Peter Müller';  
  }  
  
}
```

## View (HTML)

```
<h1>{{name}}</h1>  
  
<button (click)="chgName()">  
  Ändern  
</button>
```

# Bindungen

- Werte und Methode in Vorlagen binden
  - Mittels Ausdrucksinterpolation
    - <h1>{{name}}</h1>
    - <h1>{{getName()}}</h1>
  - Als Eigenschaft binden
    - <img [src]= "imgPath">

# Bindungen

- Werte und Methode in Vorlagen binden
  - Als Attribut binden
    - <img [attr.alt]= "imgAlt">

# Ausdrücke interpolieren

- Ausdruck in geschweiften Klammern
  - {{ AUSDRUCK }}
- Erlaubte Ausdrücke
  - Eigenschaften, Zeichenketten, Operatoren
  - Methodenrückgabe

# Eigenschaften

- Erlaubt Zuweisung über Eigenschaften eines HTML-Elementes
- [ EIGENSCHAFT ]=„AUSDRUCK“
- Erlaubte Ausdrücke
  - Eigenschaften, Zeichenketten, Operatoren
  - Methodenrückgabe

# Attribute

- Erlaubt Zuweisung über Knoten-Attribute eines HTML-Elementes
- [ attr.EIGENSCHAFT ]=„AUSDRUCK“
- Erlaubte Ausdrücke
  - Eigenschaften, Zeichenketten, Operatoren
  - Methodenrückgabe

# Styles

- Erlaubt Zuweisung über StyleEigenschaften eines HTML-Elementes
- [ style.EIGENSCHAFT.EINHEIT ]=„AUSDRUCK“
- Erlaubte Ausdrücke
  - Eigenschaften, Zeichenketten, Operatoren
  - Methodenrückgabe

# Class

- Erlaubt styling über CSS-Klassen
  - [class.KLASSENNAME]=„BOOL-AUSDRUCK“
  - [class]=„AUSDRUCK“
- Erlaubte Ausdrücke
  - Eigenschaften, Zeichenketten, Operatoren
  - Methodenrückgabe

# Ereignis

- Erlaubt Bindung von Ereignissen
  - (EVENT)=„METHODE( \$PARAM )“
- Parameter
  - \$event -> reicht Ereignis durch
- Beispiel
  - (click)=„clickHandler(\$event)

# Eltern-Kind-Kommunikation

## Eltern-Komponente

```
export class UserListComponent {  
  userList: User[]; ——————  
  selectUser (user: User) {} ←—————  
}
```

```
<user  
  [userData] = "userList[0]"  
  (onSelect) = "selectUser($event)"  
 >
```

## Kind-Komponente

```
export class UserComponent {  
  @Input() userData: User;  
  @Output() onSelect: EventEmitter;  
}
```

# Komponentenattribute

- Benutzerdefinierte Attribute lassen sich über den Eigenschaftsdekorator anlegen
  - `@Input(OPT_ATTR_NAME)` name: Type
- Auch für Setter nutzbar
- `ngOnChanges` : Hook informiert über neue Werte
  - `SimpleChanges`

# Komponentenereignisse

- Benutzerdefinierte Ereignisse lassen sich über den Eigenschaftsdekorator anlegen
  - `@Output(OPT_ATTR_NAME)` name: `EventEmitter<T>`
- `EventEmitter` sendet Wert via `emit`
- Elter-Komponenten können sich an das Ereignis hängen
  - `$event` – Übertragener Ereigniswert

# Komponenten-Lebenszyklus

constructor

ngOnChanges

ngOnInit

ngDoCheck

ngAfterContentInit

ngAfterContentChecked

ngAfterViewInit

ngAfterViewChecked

ngOnDestroy

export class UserListComponent

<userList [data] = "userList">

<userList> Vorlage

<user></user>

<user> </user>

</userList>



netTrek



NewElements

# HostBindings- und Listender

- Mittels Eigenschaftsdekorator lassen sich auch Bindungen direkt in der Komponentenklasse definieren
  - `@HostBinding (bind) NAME : boolean = true`
  - `@HostListener (EVT_NAME, [, $event']) HANDLER : Function = (evt)=>{}`

# Direktive

,

# Direktiven

- Definition
- Hauseigenen
  - ngIf
  - ngFor
  - ngClass und ngStyle
- Eigene Direktiven

# Direktiven

- Direktiven lassen sich innerhalb einer Vorlage nutzen
- Sie werden als Attribute ausgezeichnet
- Es gibt zwei Typen von Direktiven
  - Strukturelle Direktiven, die den DOM manipulieren
  - Attribut-Direktiven, die das Aussehen und/oder Verhalten eines Elements manipulieren

# Direktiven

- Strukturelle Direktiven sind durch ein Asterix (\*) vor dem Attributnamen erkennbar:
  - <img \*ngIf="showImg">
  - <li \*ngFor="let label of labels">

# Direktiven

- Attribut-Direktiven ohne Wert:
  - <input matInput>
- Attribut-Direktiven mit Wertzuweisung:
  - <textarea matAutosizeMinRows="2">
- Attribut-Direktiven mit gebundener Wertzuweisung
  - <input [ngClass]=""inputClass">

# Strukturelle Direktiven - nglf

- [ngIf]= „AUSDRUCK“
  - Hängt den Knoten aus dem DOM wenn der Ausdruck false ist

# Strukturelle Direktiven - ngFor

- [ngFor]=„AUSDRUCK“
  - Wiederholt den Knoten anhand einer Iteration
  - Ausdruck
  - Beschreibt Iterator und kann zusätzliche Werte durchreichen
    - index, first, last, even, odd

# Attribute Direktiven

- [ngClass]=„AUSDRUCK“
- [ngStyle]=„AUSDRUCK“
- Erweitert style und class Attribut eines Knotens

# Direktive erstellen

- @Directive
  - selector
    - Attribut z.B. [,myDirective']
    - Klasse z.B. ,.my-class' (auch als Liste)
  - class optional mit DI von ElementRef
    - nativeElement - Referenziert dann das Element

# Pipe

# Pipes

- Pipes dienen der Manipulation von Ausgaben
- Sie werden überwiegend in Vorlagen genutzt
  - Ausdruck | PipeName : Parameter
- Die Nutzung auf Code-Ebene ist aber auch möglich
  - DI oder new und transform Methode der Instanz

# Pipes

- Beispiel
  - <h1>{{name | uppercase}}</h1>
- Pipes lassen sich auch in Kette schalten
  - <h1>{{createdAt | date : 'long' | uppercase}}</h1>

# Pipes

- Hauseigene
  - Uppercase
  - Lowercase
  - Date
- ...

# Pipes

- Hauseigene
  - Uppercase
  - Lowercase
  - Date
- ...

# Pipes erstellen

- @Pipe
  - name: string
- class NAME implements PipeTransform
  - transform(value: any, args?: any): any {

**rxjs**

# rxjs - Observable

- Iterierbares Objekt, welches filter- und registrierbar (Subscription) ist, um async. Prozesse zu verfolgen
- Cold
  - Observable wartet auf Subscription
- Hot
  - Observable arbeitet bereits

# rxjs - Obserer

- Sendet Werte, Fehler und Status in den Datenstrom
  - next
  - error
  - complete

# rxjs - Subject

- Sowohl Observer als auch Observable
  - Damit registrierbarer Datenstrom
  - Und Sender in einem

# rxjs - Subscription

- Registrierung an Observable
  - next
  - error
  - complete
- unsubscribe (Deregistrierung)
- siehe: <http://rxmarbles.com/>

# rxjs – Erstellung eins Observables

- new
- of
- range
- fromEvent
- ...

# rxjs – Operationen am Datenstrom

- Pipe
  - map
  - filter
  - find
  - scan
  - ...

# Dependency Injection

## Service und Provide Grundlagen

# Services

- Sind View-unabhängige Logiken
  - z.B. Client-Server-Kommunikation
- Sind JavaScript-Klassen
  - Instanzbereitstellung über Dependency Injection
    - provide
    - Typisierter Parameter im Konstruktor

# Dependency Injection

- Services, Werte und Funktionen können injiziert werden
- Benötigt: Bereitstellung innerhalb eines Containers (**Injector**)
  - Bereitstellung durch Anhänge in **providers**-Liste
    - Innerhalb von Metadateninformationen für
      - Module
      - Komponenten

1

## ModulA

- Register (**declarations**)
  - KomponenteA
- Bereitstellen (**providers**)
  - ServiceA

2

## KomponenteA

```
constructor(  
    service: ServiceA  
) {
```

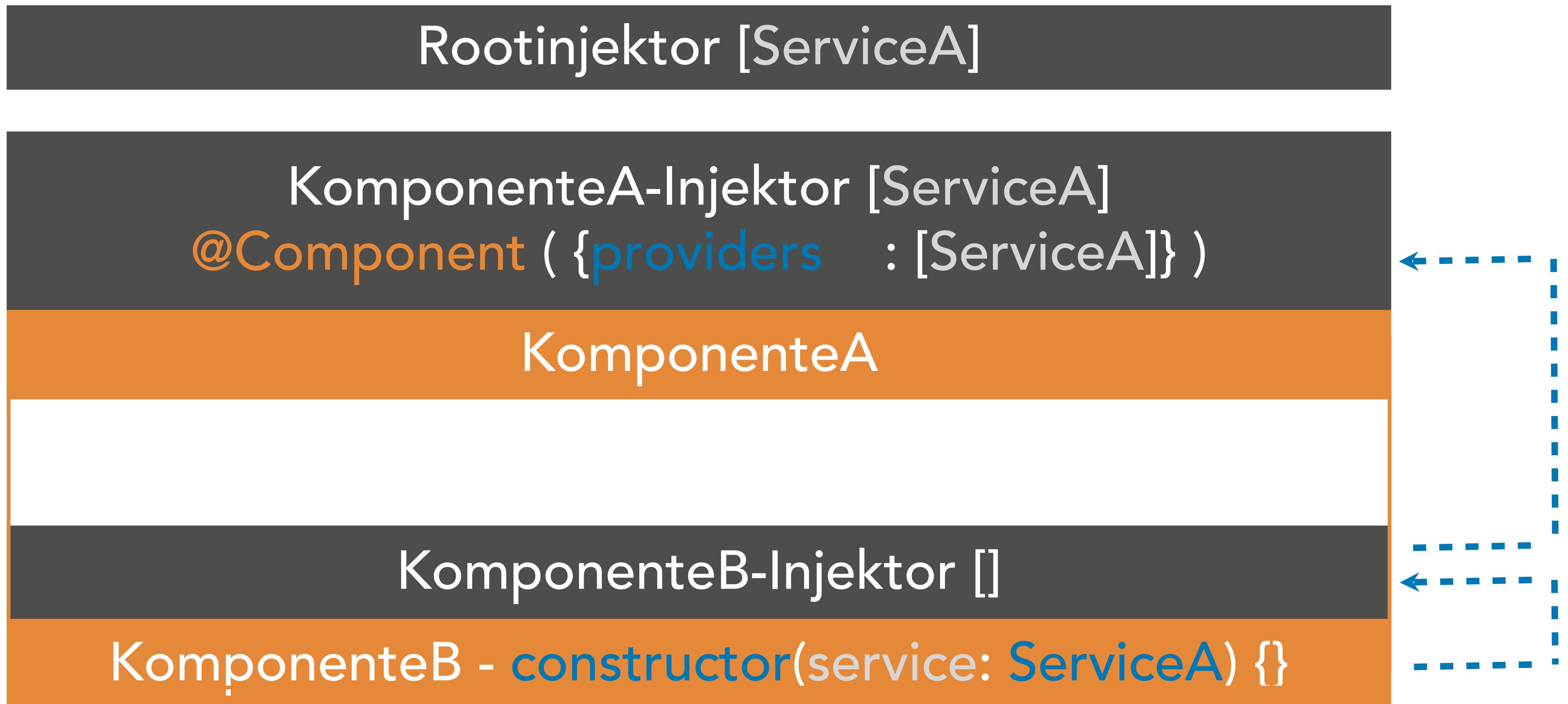
# Dependency Injection

Rootinjektor der Anwendung  
[ ServiceA ]

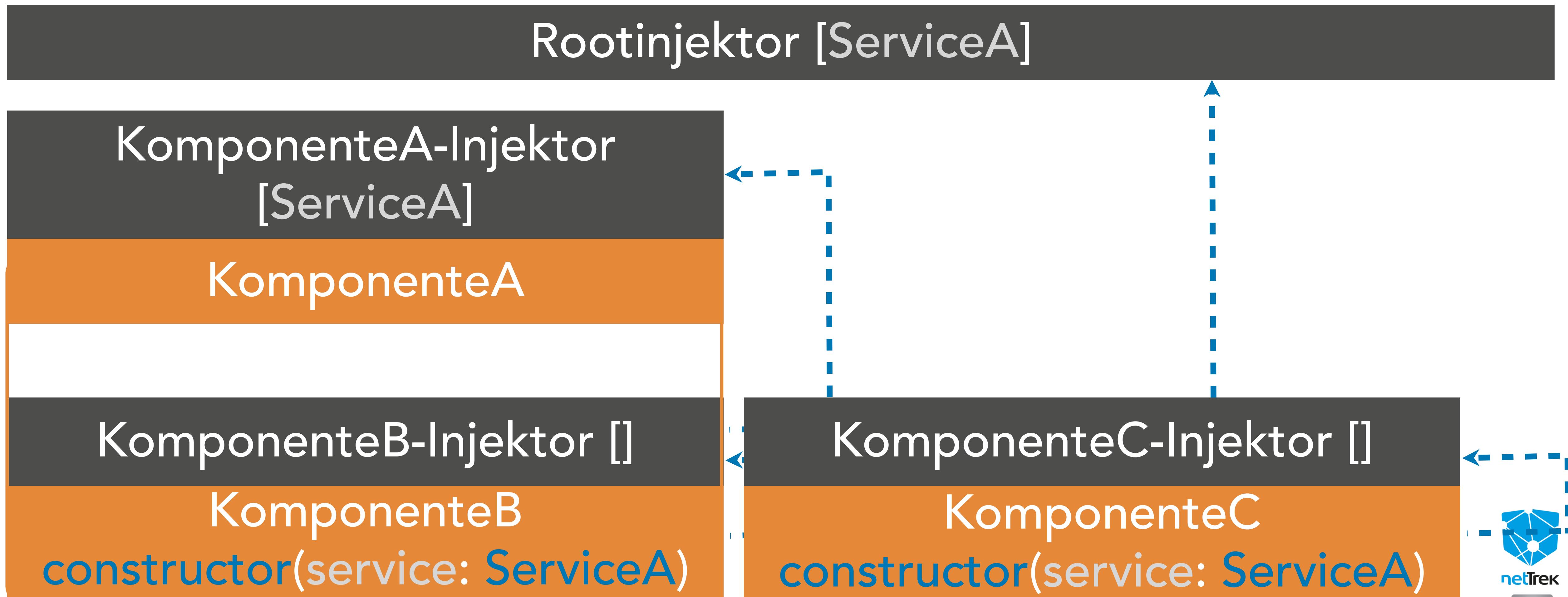
ModulA  
`@NgModule ( { providers : [ServiceA] } )`

KomponenteA - `constructor(service: ServiceA) {}`

# Dependency Injection



# Dependency Injection



# Provide von Werten im Injector

- Nutzung von **StaticProvider** Typen statt Klassen
  - **ValueProvider**
  - **ClassProvider**
  - **ExistingProvider**
  - **FactoryProvider**

# ValueProvider

- Werte im Injector registrieren
  - `provide`: any
    - Referenz zum injizieren
  - `useValue`: any –
    - Wert
  - `multi?`: boolean
    - Nutzung als Liste

# Injizierten-Wert nutzen

- Werte die im Injector bereitgestellt wurden lassen sich Injizieren
  - @Inject Decorator
  - Referenz
  - Token

# ClassProvider

- Klassen im Injector registrieren
  - Wie ValueProvider
  - `useClass: Type<any>` – statt ~~useValue~~
  - Klasse
    - sollte für aot im ES6-Modul exportiert sein

# ExistingProvider

- Existierende Werte nutzen erneut registrieren
  - Wie ValueProvider
  - `useExisting: any` – statt ~~useValue~~
  - Referenz zu einem bereits registrierten Objekt

# FactoryProvider

- FactoryMethode zum registrieren im Injector
- Wie ValueProvider
  - `useFactory`: Function – statt `useValue`
  - Factory-Methode
  - `deps`: [any]
    - Liste von Abh.

# DI-Decoratoren

- **@Injectable** – Zeichnet Service-Klassen aus, damit diese wiederum die DI im Konstruktor nutzen können
- **@Inject** - injiziert anhand eines Tokens
- **@Optional** – wir vor @Inject verwendet, ermöglichen optionale Injizierung
- **@Self, @Host, @SkipSelf** – wird genutzt, um das Injector-Bubbling zu kontrollieren

# InjectionToken

- Erzeugt Referenz-Token zu einer DI
  - Generische Type verweist auf Werte-Typ der DI

# ModuleWithProviders

- Erzeuge ein Modul
  - MetaDaten enthalten allg. Imports und Deklarationen etc.
  - Modul-Klasse verfügt über eine statische Factory
    - Rückgabe ist das Modul selbst mit konfigurierten Provides => ModuleWithProviders

# ModuleWithProviders

- `@NgModule ( {} )`

```
export class MyModule {  
    static forRoot ( config ): ModuleWithProviders {  
        return {  
            ngModule : MyModule,  
            providers: [ ... ]  
        };  
    }  
}
```

# HTTP

## CRUD via HttpClient

# HTTP-Service

- `HTTPClientModul` importieren
- `HttpClient-Service` injizieren
- Methoden
  - `request` – Basis aller anderen Methoden
    - `HttpRequest`

# HttpRequest Methoden

- method: string,
- url: string,
- init?:
  - body?: any;
  - headers?: HttpHeaders;
  - set ( key, value )

# HttpRequest Methoden – init

- params?: HttpParams;
  - set( key, value )
- reportProgress?: boolean
- responseType?:
  - 'arraybuffer' | 'blob' | 'json' | 'text';

# HTTP-Service Methoden

- [C] post
- [R] get
- [U] put
- [D] delete

# HttpInterceptor

- Beeinflusst Request global
- Service, dass **HttpInterceptor** implementiert
  - `intercept ( req: HttpRequest<any>, next: HttpHandler ): Observable<HttpEvent<any>>`
  - `return next.handle(req);`

# HttpInterceptor - provider

- provide:
  - `HTTP_INTERCEPTORS`,
- `useClass`:
  - Name of Interceptor-Service,
- `multi` :
  - `true`

# HttpInterceptor - NoCache

- *// needed für IE 11*  
intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {  
 req = req.clone({  
 setHeaders: {  
 'Cache-Control': 'no-cache',  
 Pragma : 'no-cache',  
 Expires : 'no-cache',  
 'Content-Type' : 'application/json',  
 Accept : 'application/json'  
 }  
 });  
 return next.handle(req);  
}

# HttpInterceptor - Progress & Error

- intercept ( req: HttpRequest<any>, next: HttpHandler ): Observable<HttpEvent<any>> {  
    console.log ( 'running Requests (start new)', ++numOfRunningRequests );  
    return next.handle ( req )  
        .pipe(  
            tap( ( event: HttpEvent<any> ) => {  
                if ( event instanceof HttpResponse ) {  
                    console.log ( 'running Requests (end success)', --  
                numOfRunningRequests );  
                }  
            }, ( error: any ) => {  
                if ( error instanceof HttpErrorResponse ) {  
                    console.log ( 'running Requests (end err)', --numOfRunningRequests  
                );  
                }  
            }  
        );  
    }  
}

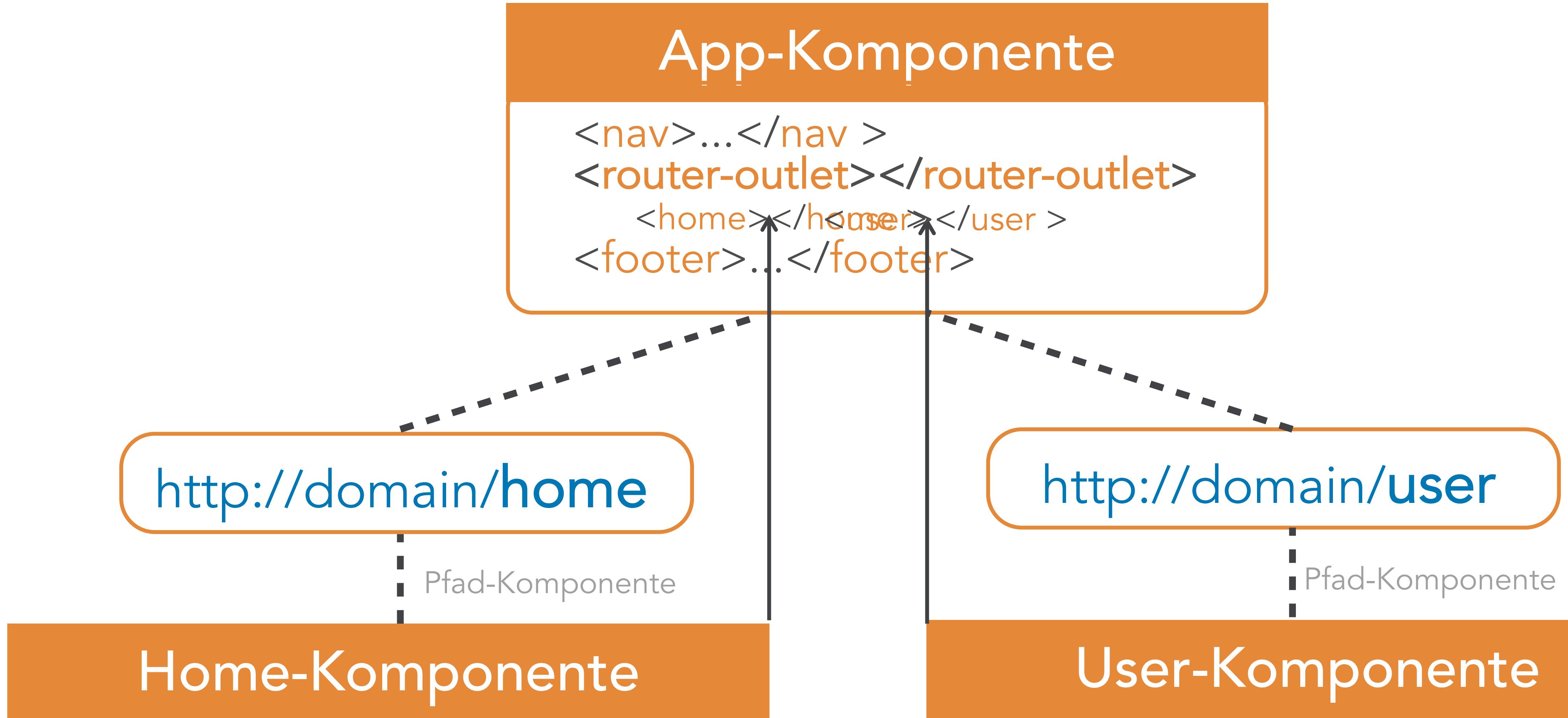
# Routing

## Basis einer SPA

# Routing

- Bestandteil des Routing-Moduls
- Basis einer Single-Page-Application
- Bestimmt, welche Komponenten bei welchem Pfad angezeigt wird

# Routing



# Modul import und Route-Def

- Modul über RouterModule.forRoot einbinden
  - Routes
    - path
    - component
  - { useHash: false }
- <router-outlet></router-outlet> einbinden

# Redirect

- initial
  - path: '',  
pathMatch: 'full',  
redirectTo: 'list'
- 404
  - path: '\*\*',  
redirectTo: 'list'

# Navigation

- **routerLink** - Directive
  - path | [ path, ...params: any[] ]
- **routerLinkActive** - Directive
  - CSS class name

# Navigation – über RouterService

- DI Router Service
- `navigate` Methode
  - Params
  - List
    - path
    - params

# Lazy Module

- `loadChildren` ermöglicht im CLI Kontext die einfache Umsetzung
- `path : 'dash',  
loadChildren : './dash/dash.module#DashModule'`
  - Der Pfad zu dem Modul und der Klassenname muss übergeben werden
  - `PFAD#MODUL_NAME`

# Lazy Module

- Im Modul selbst wird die Route mit der darzustellenden Komponente definiert
- `RouterModule.forChild ( [`  
    {  
        `path : '',`  
        `component: DashComponent`  
    }  
])

# Lazy Module

- Module Vorladen
- `RouterModule.forRoot( [], opt )`
  - `opt`
    - `enableTracing: true,`
    - `preloadStrategy: PreloadAllModules`

# Parameter

- Route mit Parameter definieren
  - `path : 'details/:id', component : UserDetailsComponent`
- In Komponente `ActivatedRoute` Service injizieren
  - `this.subscription = this.route.paramMap.pipe ( .map ( paramMap => paramMap.get ('id') || 'None' ) ) .subscribe( id => this.param_id = id );`

# Resolve-Guard

- Daten vor Routenwechsel beschaffen
  - ResolveService auf Basis des Resolve Interface anlegen, einbinden und in Route einbinden
  - `path : 'details/:id', component : UserDetailsComponent, resolve: { user: ResolveService }`

# CanActivate - Guard

- Genehmigung der Aktivierung einer neuen Route
- Hierfür wird ein auf dem **CanActive**-Interface basierender Service erstellt und eingebunden
  - `canActivate ( route : ActivatedRouteSnapshot, state : RouterStateSnapshot ) : Observable<boolean>|Promise<boolean>|boolean`

# CanActivate - Guard

- Service wird in die Routendefinition implementiert
  - path: 'home',  
component: **HomeComponent**,  
canActivate: [ **CanActiveService** ]

# Ereignisse

- Router Service injizieren
- events **Observable<Event>** subscriben
  - constructor ( router: **Router** ) {  
    router.**events**.subscribe( event => console.log (event));  
}

# Child

- Eine Route kann Unteroutes haben
- Diese müssen in der Config unter der Eigenschaft
  - `children`
  - analog zur vorhanden Konfiguration angelegt werden.

# Formulare

# Formulare

- Umsetzbar auf zwei Wege
  - Vorlagen-getrieben
    - Dabei gibt die Vorlage das Formularmodel und die Validatoren vor (ähnlich AngularJS)
  - Reaktiv (Daten-getrieben)
    - Hierbei werden die Formularelemente vorab geplant und an ein Formular in der Vorlage gebunden

# Formulare - Vorlagen-getrieben

- Vorbereitend: Einbindung des **FormsModuls** zur
- Anschließend sind Formular-Direktiven in der Vorlagen-Schicht nutzbar:
  - **ngModel, required, minlength, ...**
  - zur Bindung von Validatoren und Werten ins Formular-Model
  - All dies wird ohne zusätzliche Programmierung realisiert

# Formulare - Vorlagen-getrieben

- **ngForm** – wird genutzt, um das Formular auszuzeichnen.
- Direktive verfügt über ein **exportAs** d.h. wir können dies für einen #Hash-Id zuordnen **#myForm='ngForm'**
  - Ermöglicht den Zugriff auf Control-Eigenschaften
    - **valid, invalid, value** etc.
    - **myForm.valid**

# Formulare - Vorlagen-getrieben

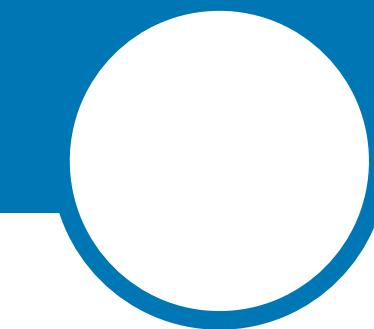
- **ngModel** kann auf drei Arten genutzt werden
  - Als Attributs-Direktive **ngModel** kombiniert mit einer Namensdefinition über das **name** Attribut.
    - Dadurch wird automatisch ein Formular-Model erzeugt
    - **myForm.value = {name: Input-Feld-Wert}**
  - Als Attributs-Direktive mit Bindung eines Initial-Wertes **[ngModel]**

# Formulare - Vorlagen-getrieben

- Vermeide: Nutzung als Attributs-Direktive mit Zweiwege-Bindung `[(ngModel)]`. Dadurch wird der Initial-Werte aktualisiert. D.h. es gibt zwei Modelle ☹
- Als Zuweisung für eine #Hash-Id z.B. `#mail='ngModel'`
  - Ermöglicht kombiniert mit der `ngModel` Direktive den Zugriff auf: `valid`, `invalid`, `value` etc.
  - `mail.valid`

# Formulare - Vorlagen-getrieben

- **ngModelGroup** Direktive zur Gruppierung von Model-Informationen
- Die Direktive muss hierarchisch in der Vorlage genutzt werden.
  - Die **input**-Knoten des Direktiven-Elementes erzeugen die Gruppenelemente.



## Form

```
<form novalidate #myForm="ngForm">  
  <input type="text"  
    autocomplete="name"  
    placeholder="name"  
    name="name" _  
    #name="ngModel"  
    ngModel  
  >  
  <span ngModelGroup="credentials">  
    <input name="email"  
      #email="ngModel" ngModel>  
    <input name="password"  
      #password="ngModel" ngModel>  
  </span>  
</form>
```



## Model

ngForm -> myForm  
ngModel -> name  
ngModelGroup -> credentials  
ngModel -> email  
ngModel -> password

## Form

```
<form novalidate #myForm="ngForm">  
  <input type="text"  
    autocomplete="name"  
    placeholder="name"  
    name="name" ——————  
    #name="ngModel"  
    ngModel  
  >  
  <span ngModelGroup="credentials">  
    <input name="email"  
      #email="ngModel" ngModel>  
    <input name="password"  
      #password="ngModel" ngModel>  
  </span>  
</form>
```

## Model

```
myForm.value = {  
  name: '...',  
  credentials {  
    email: '...',  
    password: '...',  
  }  
}
```

# Formulare – Controls-

- **ngForm** und **ngModel** – sind Control-Direktiven mit folgenden Eigenschaften:
  - **value** - Wert
  - **valid, invalid** - Valide
  - **touched, untouched** - Berührt
  - **dirty, pristine** – Benutzt/Unbenutzt
  - **errors?** – Validator-Fehler



# Formulare – Controls

- Control Methoden:
  - `setValue, reset` – Wert
  - `markAsTouched, markAsUntouched` - Berührt
  - `markAsDirty, markAsPristine` – Benutzt/Unbenutzt
  - `setErrors?` – Validator-Fehler

# Formulare – Validatoren

- Validatoren lassen sich über Direktiven einbinden
  - **required** – erforderlicher Wert
  - **email** – Gültige Mail
  - **minlength, maxlength** – Längen-Prüfung
  - **pattern** – Ausdrucks-Prüfung



netTrek



NewElements

# Formulare – Validatoren

- Validatoren legen im **errors** Objekt des Controls Fehlerinformationen in abh. zum Validator ab.
- Fehlermeldungen lassen sich entsprechend darstellen
- `<div *ngIf="email.errors?.required">...</div>`
- Das Fragezeichen bindet optionale Werte

# Formulare – Daten senden

- (submit) – Verwenden auf dem Formular das Submit-Ereignis
- Nutzen als Auslöser im Formular einen <button> oder <a> vom Typ submit
- Verwende auf dem Auslöser zusätzlich die disable-Direktiven, zum Deaktivieren bei ungültigen Formularen.

```
<form novalidate #myForm="ngForm" (submit)="send(myForm)">
```

```
<button type="submit" [disabled]="myForm.invalid">senden</button>
```



# Formulare – Daten zurücksetzen

- (reset) – Verwenden auf dem Formular das Reset-Ereignis
- Nutzen als Auslöser im Formular einen <button> oder <a> vom Typ reset
- Verwende auf dem Auslöser zusätzlich die disable-Direktiven, zum deaktivieren, wenn noch keine Formularwerte eingetragen sind Formularen.

```
<form novalidate #myForm="ngForm" (submit)="send( myForm )"  
      (reset)="reset( myForm, $event )">
```

```
<button type="reset" [disabled]="!myForm.dirty">reset</button>
```

# Formular CSS-Klassen

- Angular fügt an input-Elemente autom. CSS-Klassen, die den Status des Controls wiederspiegeln.
  - ng-untouched, ng-touched
  - ng-pristine, ng-dirty
  - ng-invalid, ng-valid

# Model-Optionen

- Die gleichnamige Direktive beeinflusst das Model-Handling
  - [ngModelOptions]="{name: 'name'}"
  - ersetzt das setzen des name-Attributes
  - [ngModelOptions]="{standalone: true}"
  - Wert wird dem übergeordneten Form nicht mitgeteilt

# Model-Optionen

- `[ngModelOptions]="{updateOn : 'blur '}"`
- Definiert einen Form-Hook (`change`, `submit`, `blur`) bei dem das Model aktualisiert werden soll.
- `debounce` - angekündigt: Update nach timeout.

# Eigene Validatoren

- Validatoren bestehen aus zwei Schichten
  - ValidatorFn
    - Funktion übernimmt die Überprüfung und reicht aus, wenn die Nutzung ausschließlich reaktiv ist.
  - Direktive
    - Nutzt die zuvor beschrieben Funktion und ist für vorlagengetriebene Formulare notwendig

# Validator-Funktion

- Funktion wird über eine Factory erzeugt, welche optional die Prüfungsbedingung entgegennimmt.
- **ValidatorFn** - erwartet: **AbstractControl**
  - gibt ein Fehlerobjekt (**ValidationErrors**) oder null zurück

# Validator-Funktion

```
export class EqualValidator {  
    static isEqual ( compare: any ): ValidatorFn {  
        return ( control: AbstractControl ): ValidationErrors | null => {  
            if ( control.value === null || compare === null ) return null;  
            return compare !== control.value ?  
                { 'equal': { 'is': control.value, 'should': compare } } : null;};  
    }  
}
```

# Validator-Direktive

- Validierung wird in `NG_VALIDATORS` durch eine neue Direktive erweitert
- Erweiterung wird im Injector der Direktive bereitgestellt.
- Direktive muss das `Validator` Interface implementieren.
  - `validate ( c: AbstractControl ): ValidationErrors | null`
    - Wird zur Prüfung ausgeführt und gibt Fehlerobjekt zurück

# Validator-Direktive

- `registerOnValidatorChange (fn: any): void;`
- Methoden übermitteln eine Referenz zu den, `onChange` Callback
- Notwendig, wenn Prüfungsbedingungen sich ändern

# Validator-Direktive – Injector erweitern

- Erweitere den `NG_VALIDATORS`
- Benutze hierfür den `ExistingProvider`
- Als Provide-Token nutze: `NG_VALIDATORS`
- Der Wert wird über `useExisting` auf die Direktiven-Klasse gesetzt, da diese nicht unmittelbar im Injector existiert wird die `forwardRef` Methode genutzt
- `multi: true` erweitert die `NG_VALIDATORS` Liste

# Validator-Direktive – Injector

- ```
export const EQUAL_VALIDATOR = {  
  provide: NG_VALIDATORS, multi: true,  
  useExisting: forwardRef(() => MyDirective) };
```
- ```
@Directive ( { selector: ,[equalValidator][ngModel]',  
  providers: [EQUAL_VALIDATOR] } )
```
- ```
export class MyDirective implements Validator {
```

# Werte Zugriff

- Falls Werte manipuliert werden müssen, bevor sie im Model oder der View genutzt werden
- Zugriff-Steuerung: `NG_VALUE_ACCESSOR` durch neue Direktive erweitern
- Erweiterung im Injector der Direktive bereitstellen.
- Direktive muss das `ControlValueAccessor` Interface implementieren.

# Werte Zugriff – Interface Methoden

- `writeValue(obj: any): void;`
  - Aufgerufen bei Modeländerungen von Form-API. Methode muss View anpassen.
- `registerOnChange & registerOnTouched (fn: any): void;`
  - Methoden übermitteln eine Referenz zu den, onChange und onTouched Callback

# Werte Zugriff – Interface Methoden

- **onChange (value): void;**
  - Callback wird aufgerufen beim Change-Ereignis
  - Übermittel wird der aktuelle Wert aus der UI.
  - Callback aktualisiert das Model über die Form-API

# Werte Zugriff – Interface Methoden

- `onTouched (): void;`
- Callback aufrufen, wenn der Status des Controls geändert werden muss
- Status gibt wieder, ob das Formular-Element aktiviert(focus/blur reicht) wurde.

# Werte Zugriff – Injector erweitern

- Erweitere den `NG_VALUE_ACCESSOR`
- Benutze hierfür den `ExistingProvider`
- Als Provide-Token nutze: `NG_VALUE_ACCESSOR`
- Der Wert wird über `useExisting` auf die Direktiven-Klasse gesetzt, da diese nicht unmittelbar im Injector existiert wird die `forwardRef` Methode genutzt
- `multi: true` erweitert die `NG_VALUE_ACCESSOR` Liste

# Werte Zugriff – Injector erweitern

- ```
export const CONTROL_VALUE_ACCESSOR = {  
  name: 'formatterParserValueAccessor',  
  provide: NG_VALUE_ACCESSOR, multi: true,  
  useExisting: forwardRef(() => MyDirective) };
```
- ```
@Directive ( { selector: 'input[msgFormater]',  
  providers: [CONTROL_VALUE_ACCESSOR] } )
```
- ```
export class MyDirective implements ControlValueAccessor {
```



# Reaktive Formulare

- Im Gegensatz zu Vorlagen-getriebenen Formularen vermeiden wir Direktiven wie: `ngModel`, `required`, `minlength`
- Statt dessen werden zuvor Controls erzeugt und anschließend in der Vorlage gebunden via:
  - `formGroup`, `FormControl`, `FormControlName` ...
  - Als Vorbereitung muss das `ReactiveFormsModule` eingebunden werden.

# Reaktive Formulare – Model erzeugen

- Erzeuge Controls für Werten über **FormControl**
  - Konstruktor erwartet **Wert** und **Validatoren**
- Erzeuge Gruppen von Werten über **FormGroup**
  - Konstruktor erwartet ein **Key-Value-Pair Objekt**
    - **Key:** Name des Controls oder der Untergruppe
    - **Value:** Instanz des Controls oder der Untergruppe



# Reaktive Formulare – Direktiven

- **[formGroup]** – Bindet die unterste Wert-Gruppe
- **formGroupName** – Bindet Untergruppe anhand des Names, das im Key-Value-Pair Objekt definiert wurde.
- **formControlName** – Bindet Control anhand des Names, das im Key-Value-Pair Objekt definiert wurde.
- **[formControl]** – Bindet eine Control-Instanz.

# Form

```
<form novalidate [FormGroup]="myForm">  
  <input type="text"  
    formControlName="name"  
  >  
  <span formGroupName="credentials">  
    <input type="email"  
      formControlName="email">  
    <input type="password"  
      formControlName="password">  
  </span>  
</form>
```

# Model

```
this.myForm = new FormGroup ( {  
  name: new FormControl ( 'Saban',  
    Validators.required ),  
  credentials: new FormGroup ( {  
    email : new FormControl (  
      'us@netTrek.de',  
      [ Validators.email,  
        Validators.required ] ),  
    password: new FormControl ( ... ) } );
```

# Form

```
<form novalidate [FormGroup]="myForm">  
  <input type="text"  
    formControlName="name"  
  >  
  <span formGroupName="credentials">  
    <input type="email"  
      formControlName="email"  
    <input type="password"  
      formControlName="password">  
  </span>  
</form>
```

# Model

```
myForm.value = {  
  name: '...',  
  credentials {  
    email: '...',  
    password: '...',  
  }  
}
```

# Formulare – Helfer – FormBuilder

- FormBuilder (DI) –Service vereinfacht die Model Erstellung und den Umgang mit FormControl und FormGroup
  - Anstelle von `new FormGroup()` nutzen wir die `group` Methode vom FormBuilder und übergeben ein Key-Value Objekt.
  - Key: Name des Controls oder der Untergruppe
  - Value: Eigenschafts-Array oder Untergruppe via `group` Methode

# Formulare – Helfer – FormBuilder

- **Value**: Eigenschafts-Array
  - Erstes Element – Startwert
  - Zweites Element: Validator oder Validator-Array

# Formulare – Helper – FormBuilder

```
this.myForm = this.fb.group({  
    name: [ 'Saban', Validators.required ],  
    credentials: this.fb.group ( {  
        email: ['us@netTrek.de', [ Validators.email,  
            Validators.required ]],  
        password: ['test1234', Validators.required ]  
    })  
});
```

# Formulare – Helfer – Control

- **get:** Methode gibt ein Control aus dem Model zurück
  - Parameter:
    - Name des Controls
    - oder Pfad (Names-Array) zu einem Control
  - `this.myForm.get( ['credentials', 'email'] ) as FormControl;`

# Formulare – Helfer – Control - Fehler

- **hasError** : Methode gibt ein Boolean zurück, ob ein bestimmter Validator-Fehler existiert
- Parameter:
  - Name des Errors z.B. **required**, **email** ...
  - Name des Controls oder Pfad (Names-Array) zu einem Control

# Formulare – Helfer – Control - Werte

- `setValue( value: any, opts?): void;`
  - `onlySelf?` : boolean [default: false]
    - Validation nur auf Control nicht auf Eltern-Komponente
  - `emitEvent?` : boolean [default: true]
    - `valueChanges` Event wird vom Control gefeuert

# Formulare – Helfer – Control - Werte

- `setValue( value: any, opts?): void;`
- `emitModelToViewChange?: boolean`
  - View wird via `onChange` über die Änderung informiert
- `emitViewToModelChange?: boolean`
  - Model wird via `ngModelChange` über die Änderung informiert

# Formulare – Helfer – Control - Werte

- `reset( value, opts?: { onlySelf?: boolean;  
emitEvent?: boolean; })`
- Setzt Control zurück
  - `value` = null oder Wert
  - Zustand wird auf `pristine` & `untouched` gesetzt

# Formulare – Helfer – Control - Status

- `markAsTouched( opts?: { onlySelf?: boolean; }): void;`
- `markAsUntouched( opts?: { onlySelf?: boolean; }): void;`
- `markAsDirty( opts?: { onlySelf?: boolean; }): void;`
- `markAsPristine( opts?: { onlySelf?: boolean; }): void;`
- `disable(opts?: { onlySelf?: boolean; emitEvent?: boolean; })`
- `enable (opts?: { onlySelf?: boolean; emitEvent?: boolean; })`

# Animation

# Animationen

- Animationen bereichern moderne Webanwendungen und können sie benutzerfreundlicher gestalten
- Angular unterstützt den Entwickler bei der Einbindung von Animationen
  - Hierfür muss das Angular-Modul eingebunden sein
    - Dieses unterstützt Animationen über die ‚Web Animations API‘

# Animationen

- Polyfills unterstützen ältere Browser zur Unterstützung der ‚Web Animations API‘
- Animationen werden in Angular über Zustände geregelt
- Eine Element kann z. B. den Zustand **active** oder **inactive** haben
- Innerhalb der Metainformationen können Animationen für den Zustandswechsel definiert werden

# Animation - Vorbereitung

- Import des `BrowserAnimationsModule`
- Erweitere die MetaDaten der Komponente um die `animations` Eigenschaft.
- Eigenschaft erwartet eine Liste: `AnimationTriggerMetadata`
  - Metadaten für Animations-Auslöser
    - über `trigger` erzeugt werden

# Animation - trigger

- Die **trigger** Methode erwartet einen
- Namen für den Auslöser
- eine Liste von **AnimationMetadata**
  - Statusdefinitionen **state**
  - Übergangssequenz **transition**

# Animation - State

- Die **state** Methode erwartet einen
  - Namen für den Status
  - AnimationStyleMetadata
    - erzeugt über die Methode **style**
      - erwartet ein Objekt mit Style-Properties
  - Optional ein Params Objekt mit Key-Value Pairs

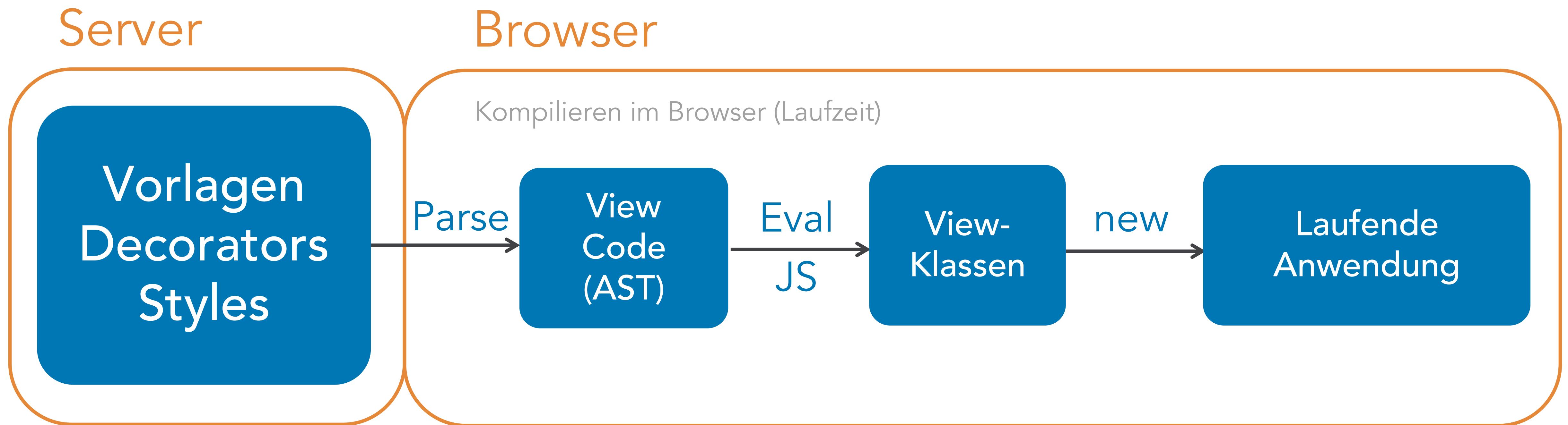
# Animation - State

- Die **state** Methode erwartet einen
  - Namen für den Status
  - AnimationStyleMetadata
    - erzeugt über die Methode **style**
      - erwartet ein Objekt mit Style-Properties
  - Optional ein Params Objekt mit Key-Value Pairs

# Veröffentlichen

## JIT, AOT und mehr

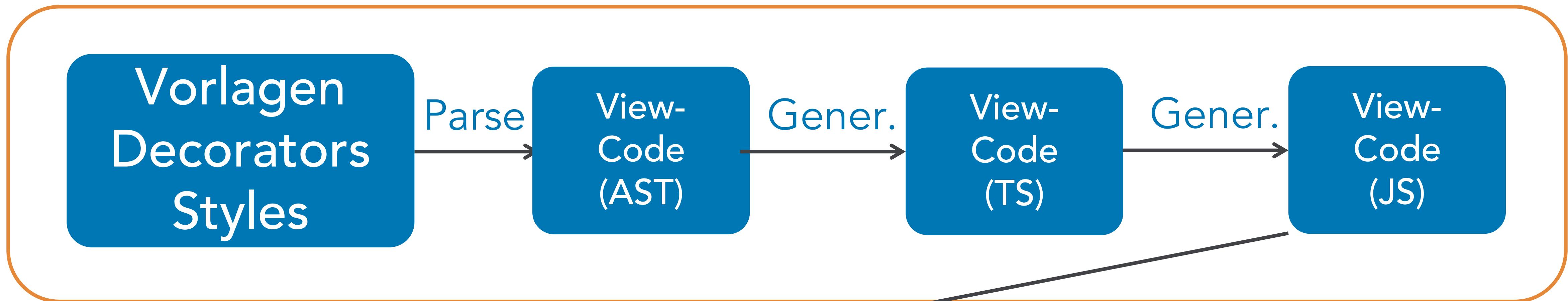
# JIT



# AOT

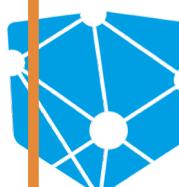
Entwickler

Angular Compiler - vorkompilieren



Server

Browser



# Veröffentlichen (cli < 6)

- ng build
- --target (-t) production | development
  - --dev (-t=development e=dev)
  - --prod (-t=production e=prod)
- --environment (-e) prod | dev <custom>
- --aot

# Veröffentlichen

Option	--dev	--prod
--aot	false	true
--environment	dev	prod
--output-hashing	media	all
--sourcemaps	true	false
--extract-css	false	true
--named-chunks	true	false
--build-optimizer	false	true with AOT and Angular 5

# Veröffentlichen (cli >= 6)

- ng build [app-name]
  - --prod
    - configuration (production)
    - --aot = true
    - --build-optimizer (true)
      - optimization (UglifyJS)

# Veröffentlichen (Environment)

- angular.json
  - configurations Bereich der App erweitern (neuerName)
  - fileReplacement definieren
- ng build [app-name]
  - configuration
    - neuerName

# Veröffentlichen – Packet Analyse

- ng build [app-name]
  - --stats-json
    - # webpack-bundle-analyzer

# ServiceWorker

## Progressiv Web App

# PWA - Grundlagen

- Webanwendung
  - Herunterladbar und installierbar
    - Updates
      - Im Hintergrund
      - Inkrementell
    - Deutlich bessere Startup-Zeit

# PWA - Anwendung erstellen

- ~~ng new angular-pwa-app --service-worker~~

- ng add @angular/pwa
  - installiert
    - @angular/pwa
    - @angular/service-worker

# PWA – angular.json Anpassungen

- angular.json
- assets
  - Einbindung der Mainfest Datei
- production Konfiguration
  - serviceWorker aktivieren [true]
  - *"ngswConfigPath": "src/ngsw-config.json"*

# PWA – Dateien

- Automatisch erzeugte Dateien bei der Initialisierung
  - `ngsw-config.json`
    - Konfiguriert welche Dateien offline Verfügbar werden

# PWA – Dateien

- Automatisch erzeugte Dateien bei der Veröffentlichung
  - `ngsw-worker.js`
    - Angular ServiceWorker – steuert PWA
  - `ngsw.json`
    - Konfiguration des ServiceWorkers

# PWA – ServiceWorker Modul

- Modul für den Serviceworker in App-Modul importieren
- ServiceWorkerModule

```
.register('/ngsw-worker.js',
  { enabled: environment.production }
)
```

# PWA – index vorbereiten

- index.html
  - muss Manifest laden
    - `<link rel="manifest" href="manifest.json">`
    - Die Mainfest beschreibt die Anwendung und sagt aus, welche Icons und Einstellungen genutzt werden.
  - Theme Farbe definieren
    - `<meta name="theme-color" content="#1976d2">`

# PWA - Anwendung testen

- `npm install -g http-server`
- `cd dist`
- `http-server -p 8080 -c1`

# PWA – SwUpdate Service

- available: Observable<UpdateAvailableEvent>
- activated: Observable<UpdateActivatedEvent>
- isEnabled: boolean;
- checkForUpdate(): Promise<void>;
- activateUpdate(): Promise<void>;

# Debugging

# Debugging

- Möglichkeiten im Browser
- Browsererweiterungen nutzen
- Debuggen mit Augury

# Performance und Architektur

## ChangeDetection & Rendering

# Komponenten und Direktiven Ökosystem

- Jede Komponente gehört zu einer ViewRef
  - Block von Elementen => Benutzeroberfläche
  - Basis für die Änderungserkennung.
- Komponenten und Direktiven DI-Eigenschaften
  - ElementRef
    - Element-Referenz (Zugang zum nativeElement)

# Komponenten und Direktiven Ökosystem

- Komponenten und Direktiven DI-Eigenschaften
  - **ViewContainerRef**
    - Referenz zum Container für dynamische Inhalte, welche hinter dem Element angehängt werden <router-outlet>
    - Embedded view – über Vorlagen-Referenzen
    - Host View – über Komponenten-Referenzen

# Eltern Kind Beziehung

- Wiederholung der wichtigsten Dekoratoren
  - `@ContentChild` & `@ContentChildren`
  - `@ViewChild` & `@ViewChildren`
  - `@Input` & `@Output`
  - `@HostListener` & `@HostBinding`



# Inhalte transkludieren

- Komponenten stellen eine View dar.
- Beschrieben wird die View in der HTML-Vorlage
- Innerhalb der Vorlage können Kinds-Komponenten mit Inhalts-Knoten versehen werden
- `<user-list>`  
`<user-item>name</user-item>`  
`</user-list>`

# Inhalte transkludieren

- Inhalts-Knoten werden transkludiert, wenn Vorlagen die **ng-content** Direktiven (Knoten) nutzen.
  - Der Knoten stellt dabei einen Platzhalter dar
  - Mittels select Attribut lässt sich definieren, für welchen Inhalt der Platzhalter greifen soll

1

## View

```
<user-list>  
  <user-header></...><br/>  
  <user-item></...><br/>  
  <user-item></...><br/>  
  <user-item></...><br/>  
</user-list>
```

2

## UserList - Template

```
<h3>user-list</h3>  
  
<ng-template><user-header></...><br/>  
  <select="user-header"><br/>  
    <user-item></...><br/>  
  </ng-content><br/>  
  <user-item></...><br/>  
<ng-template></ng-content>
```

# Zugriff auf transkludierte Inhalte

- Über Eigenschafts-Decorator
  - `@ContentChild`
    - Parameter: Komponentenklasse
    - Optionales Options-Objekt mit Read-Eigenschaft
      - `{ read:`
      - `ElementRef | ViewContainerRef | Directive | Service }`

# Zugriff auf transkludierte Inhalte

- Über Eigenschafts-Dekorator
- Zugriff erst nach Hook
  - `ngAfterContentInit`

# Zugriff auf transkludierte Inhalte

- Über Eigenschafts-Dekorator
  - @ContentChildren
    - Parameter: Komponentenklasse
    - Erzeugt
      - QueryList<Type>
        - changes -> Observable

# Vorlagen Elemente ermitteln

- Über Eigenschafts-Decorator
  - @ViewChild
    - Parameter: Komponentenklasse / Hash-ID
    - Optionales Options-Objekt mit Read-Eigenschaft
      - { read:  
ElementRef | ViewContainerRef | Directive | Service }

# Vorlagen Elemente ermitteln

- Über Eigenschafts-Dekorator
- Zugriff erst nach Hook
  - `ngAfterViewInit`

# Vorlagen Elemente ermitteln

- Über Eigenschafts-Dekorator
  - @ViewChildren
    - Parameter: Komponentenklasse
    - Erzeugt
      - QueryList<Type>
        - changes -> Observable

# Dynamische Inhalte erstellen

- Vorlagen dynamisch nutzen
  - Strukturelle Direktive: `*ngTemplateOutlet`
    - Vorlagen-Referenz
    - `ViewContainerRef` (via DI)
      - `createEmbeddedView` - Methode
        - `templateRef : TemplateRef<T>`



# Dynamische Inhalte erstellen

- Eigene Strukturelle-Direktiven lassen sich als Vorlagen-Referenz nutzen
  - via `@ViewChild` oder `@ContentChild` kann die Direktive ermittelt werden
  - mittels `read` Eigenschaft kann die Template Referenz der Direktive ermittelt werden
  - `@ContentChild ( UserHeaderDirective, { read: TemplateRef } )`

# Dynamische Komponenten erstellen

- Komponente in `entryComponents` aufnehmen
  - Strukturelle Direktive: `*ngComponentOutlet`
    - Ausdruck zur Bindung einer Komponenten-Klasse
  - Optionale Eigenschaften
    - `ngOutletInjector`, `ngOutletProviders` & `ngOutletContent`

# Dynamische Komponenten erstellen

- Komponenten Factory erzeugen
  - ComponentFactoryResolver (DI)
    - resolveComponentFactory Methode
  - Parameter: Komponente  
(muss in entryComponents sein)

# Dynamische Komponenten erstellen

- Dyn. Komponente über Factory anhängen
- ViewContainerRef (via DI)
  - createComponent - Methode
  - compFactory : ComponentFactory<T>

# Change Detection

# Änderungserkennung

- Wird bei jedem asynchronen Ereignis ausgeführt
- Dabei erfolgt die Erkennung in Kette
  - Eltern-View -> Kinds-View
- Hooks werden vor (init, change, doCheck) und nach (after) der Erkennung ausgeführt
- DOM Aktualisierung ist die Folge der Änderungserkennung

# Änderungserkennung: Unter der Haube

- Die **ChangeDetectionRef** ist die Super-Klasse der **ViewRef**
- Jede Direktive und Komponente ist Bestandteil einer View
- Die View steuert somit die Änderungserkennung und DOM Aktualisierung
- Die **ChangeDetectionRef** lässt sich via DI injizieren

# Änderungserkennung - Beeinflussen

- ChangeDetectionRef
  - detach()
    - Entfernt Komponente von der Änderungserkennung
    - Analog - ChangeDetectionStrategy.OnPush
  - reattach()
    - Hängt Komponente wieder in die Erkennung

# Änderungserkennung - Beeinflussen

- ChangeDetectionRef
  - markForCheck()
    - selbst -> Kinder
  - detectChanges()
    - Erkennung Eltern -> Kindern

# Änderungserkennung - beeinflussen

- Änderungserkennung umgehen
  - *ngZone (DI)*
  - *runOutsideAngular*

# Änderungserkennung - OnPush

- Änderungserkennung mit der Push Strategie
  - Wird durch asynchrone Ereignisse ausgelöst
  - Wird bei Wertwechsel einer Bindung ausgelöst
  - *async Pipe* – Unterstützt bei der Auslösung

# Unit-Testing

# Grundlagen

- Setup
  - angular.json
  - projects > [name] > architect > test
    - main > **test.ts**
  - tsConfig > src/tsconfig.spec.json
  - karmaConfig > src/karma.conf.js

# Grundlagen

- test.ts
  - Einstellung für TestBed (Test-Umgebung)
    - JIT Setup
    - Spec Definitionen

# Grundlagen

- tsConfig
  - TypeScript Compiler Einstellungen
    - module > commonjs
- karmaConfig
  - Test-Runner für Unit Tests z.B. mit Jasmine
  - Setup des Karma-Umgebung

# Grundlagen

- Setup des Karma-Umgebung
  - Pfade und Frameworks
    - Jasmine & @angular-devkit/build-angular
  - Setup der Plugins
    - Framework, Reporter und Launcher
  - Setup generelle Setups

# Spec Dateien

- describe Block für einen Test erstellen
  - description: string
  - callback handler
- describe callback hat zwei Haupt-Phasen
  - beforeEach (jasmine) zu vorbereiten der Test-Assets
  - it (jasmine) zum testen

# Spec Dateien

- `beforeEach` (jasmine) erwartet ein callback handler, der Test-Assets vorbereitet
- `it` (jasmine function) erwartet zwei Parameter
  - `description: string` -> dargestellt während der Tests
  - callback handler führt die Tests aus

# Spec Dateien

- callback - body
  - Innerhalb des Handlers werden die Tests ausgeführt über
  - `expect (jasmine)` Methode
  - parameter – zu testender Wert
  - return Instanz zur Prüfung der Übereinstimmung

# Spec Dateien

- Test der Übereinstimmung
  - `toBe (val)` -> vergleichbar `==`
  - `not.toBe(val)` -> vergleichbar `!=`
- `toEqual(val)` -> vergleicht Objekt und alle Felder
- `toMatch(regExp)` -> vergleichbar `regExp`
- `toBeDefined (val)` -> vergleichbar `!= undefined`



# Spec Dateien

- Test der Übereinstimmung
  - **toBeUndefined (val)** -> vergleichbar === undefined
  - **toBeNull (val)** -> vergleichbar === null
  - **toBeTruthy(val)** -> vergleichbar === Boolean(val)
  - **toBeFalsy (val)** -> vergleichbar !== Boolean(val)
  - **toContain (val)** -> vergleichbar indexOf !== -1



# Spec Dateien

- Test der Übereinstimmung
  - `toBeLessThan (val)` -> vergleichbar < val
  - `toBeGreaterThan (val)` -> vergleichbar > val

# TestBed

- Angular test utils
  - configureTestingModule Factory für Testing Modules
  - Einsatz vor jedem Test im Ruhezustand
  - als Parameter wird ein NgModule MetaData-Objekt übergeben
  - compileComponents() - kompiliert alle Komponenten im Module zu Inline JavaScript

# TestBed

```
beforeEach(async(() => {
  TestBed.configureTestingModule({
    imports: [
      RouterTestingModule
    ],
    declarations: [
      AppComponent
    ],
  }).compileComponents();
}));
```

# TestBed

- `createComponent( Komponenten Klasse)` Methode schließt die TestBed-Konfiguration und gibt eine `ComponentFixture` Instanz zurück.
- bietet Zugriff auf ein Debug-Element, das die Instanz der Komponente enthält.

`fixture.debugElement.componentInstance`

# TestBed

- `detectChanges()` Methodw der `ComponentFixture` Instanz führt die Änderungserkennung aus & rendert das Template
- Das Fixture Debug-Element bietet das `nativeElement` der Komponente  
`fixture.debugElement.nativeElement`
- Das `nativeElement` unterstützt `querySelector`. Alternativ kann `query`, vom `debugElement`, kombiniert mit der `By` Helper-Klasse verwendet werden, um ein Element innerhalb der Komponente auszuwählen.

# TestBed

```
expect(fixture.debugElement.nativeElement  
    .querySelector('h1').textContent)  
    .toContain('Welcome to post!');  
  
expect(fixture.debugElement  
    .query(By.css('h1'))  
    .nativeElement.textContent)  
    .toContain('Welcome to post!');
```

# Helper - async

```
let fixture: ComponentFixture<AppComponent>;
let app: AppComponent;
let h1DebugElement: DebugElement;
let h1: HTMLElement;

beforeEach ( async () => {
  TestBed.configureTestingModule ( {
    imports      : [...],
    declarations: [ AppComponent ]
  } ).compileComponents ();
} );

beforeEach ( () => {
  fixture = TestBed.createComponent ( AppComponent );
  app     = fixture.debugElement.componentInstance;
  h1DebugElement = fixture.debugElement.query ( By.css ( 'h1' ) );
  h1 = h1DebugElement.nativeElement;
} );
```

# Helper - inject

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

it ( 'should be created' ,
  inject ( [ UserResolveService ] ,
    ( service: UserResolveService ) => {
      expect ( service )
        .toBeTruthy ();
    } ) );
}
```

# Helper - HttpClientTestingModule

- Verwende das Modul als Abh. in TestBed
- Nutze den HttpTestingController zum mocken
- Erzeuge einen **TestRequest** mit dem Ctrl.
  - `testRequest = httpMock.expectOne( url )`
  - sende Response mittels **flush** Methode.
    - `testRequest.flush( body, opts? );`

# Helper - HttpClientTestingModule

- Im afterEach nicht genutzte Requests entfernen
- `afterEach( () => {  
 httpMock.verify();  
});`

# e2e-Testing

# Protractor

- Blackbox e2e Tests
- Entwickelt von Google aus Basis
  - Selenium
  - Webdriver
- kein Ersatz für Unit-Tests
  - sehr gute Ergänzung

# Protractor - Konfiguration

- Anular.json
  - Eigenes Projekt für e2e
    - protractorConfig
    - devServerTarget
    - tsConfig

# Protractor - ProtractorConfig

- Einstellungsmöglichkeiten für
  - Test-Framework **jasmine**
  - Browser **jasmine**
    - URL
  - Tests **./src/\*\*/\*.e2e-spec.ts**
  - tsConfig **tsconfig.e2e.json**
  - uvm.

# Protractor - Tests

- Vergleichbar Unit-Tests
  - öffnen einer Seite `browser.get('/');`
  - Ausführen von Tests mit `jasmine`
    - `expect(page.getParagraphText()).toEqual('Welcome!');`

# Protractor - Helfer

- aus dem Protractor Modul
  - browser: ProtractorBrowser
    - get('/');
    - getTitle()

# Protractor - Helfer

- element (by-statement): ElementFinder
  - all(by.repeater('result in memory'));
- by: ProtractorBy
  - css( selector );

# Protractor - Helfer

- element: WebElement
  - click( );
  - sendKeys( keys/string [] )
  - getCssValue( cssProp )
  - getText()
  - isEnabled()

# Protractor - Helfer

- element: WebElement
  - isSelected()
  - isDisplayed()
  - submit()