

Angular 8.x

Saban Ünlü

Zwei Worte zu mir

Saban Ünlü

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



Projektsetup

Erste Schritte

- Mac
 - XCODE installieren
 - node.js installieren ($\geq 10.9.x$)
- Win
 - node.js installieren ($\geq 10.9.x$)
 - Git installieren (inkl. Bash)

npm Proxy ?

- `npm config set proxy http://PROXYURL`
- `npm config set https-proxy https://PROXYURL`

Setup Manuell

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

angular-cli

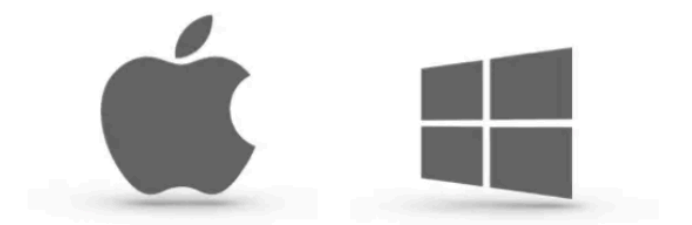
- Kommandozeilen Tool
 - Initialisieren & einrichten
 - Entwickeln und Warten
 - Testen und veröffentlichen

angular-cli - installieren

- `npm install -g @angular/cli`
- optional
 - Entwickeln und Warten
 - CLI im App Kontext



Angular Console



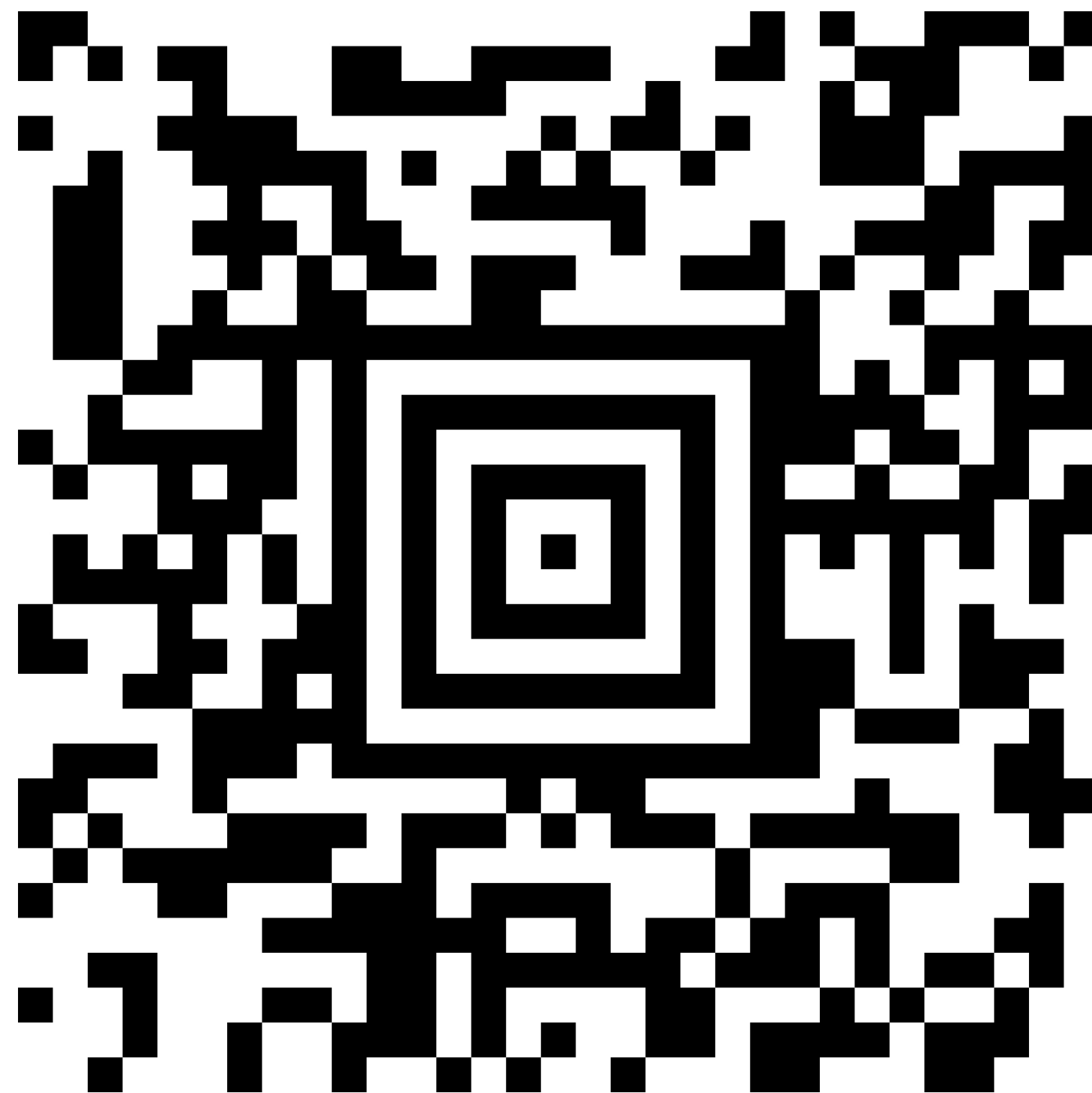
angular-cli

- ng new netTrek --prefix=nt
- ng serve
- ng g server --aot
- ng build
- ng build --prod
- ng lint
- ng test
- ng e2e



Trainings Branch

- `git clone -b training/msg19`
`https://github.com/netTrek/ng8Adv.git msg19`



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

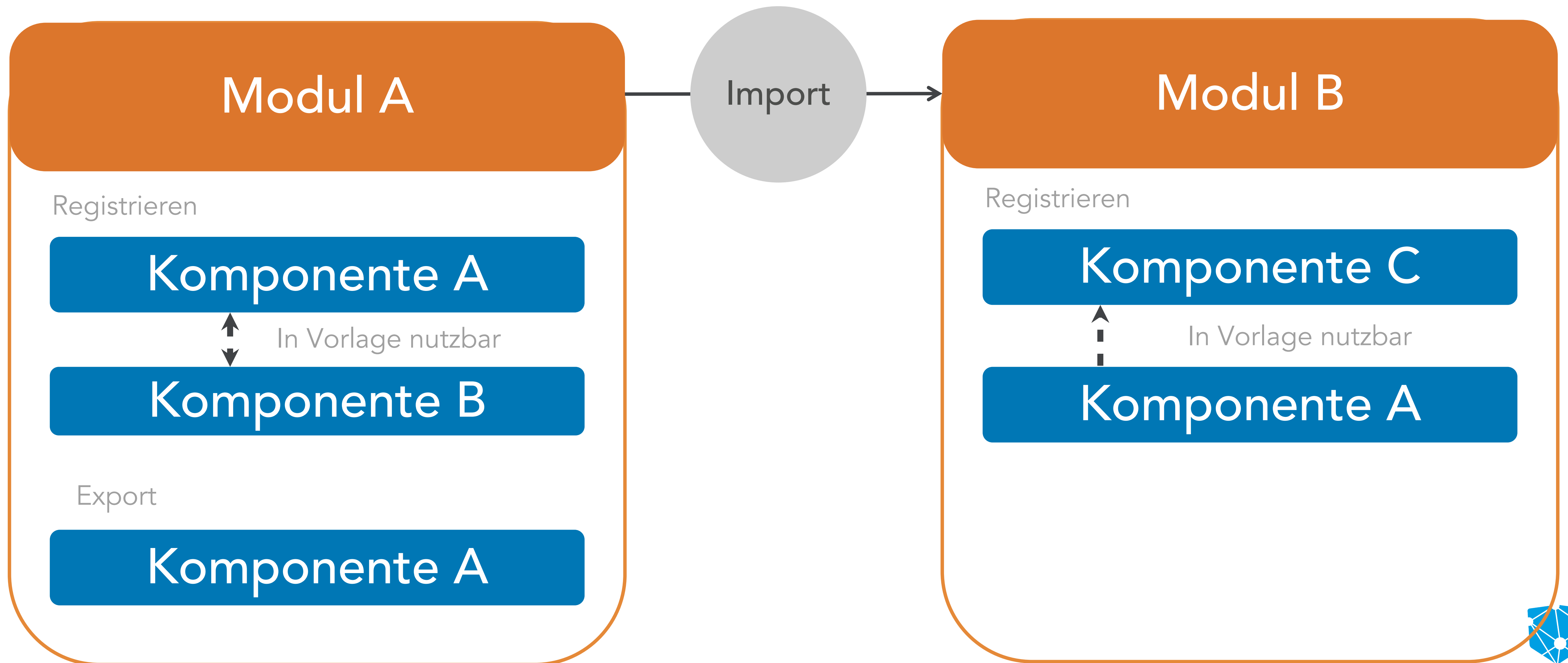
Architektur

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 user --module app` 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

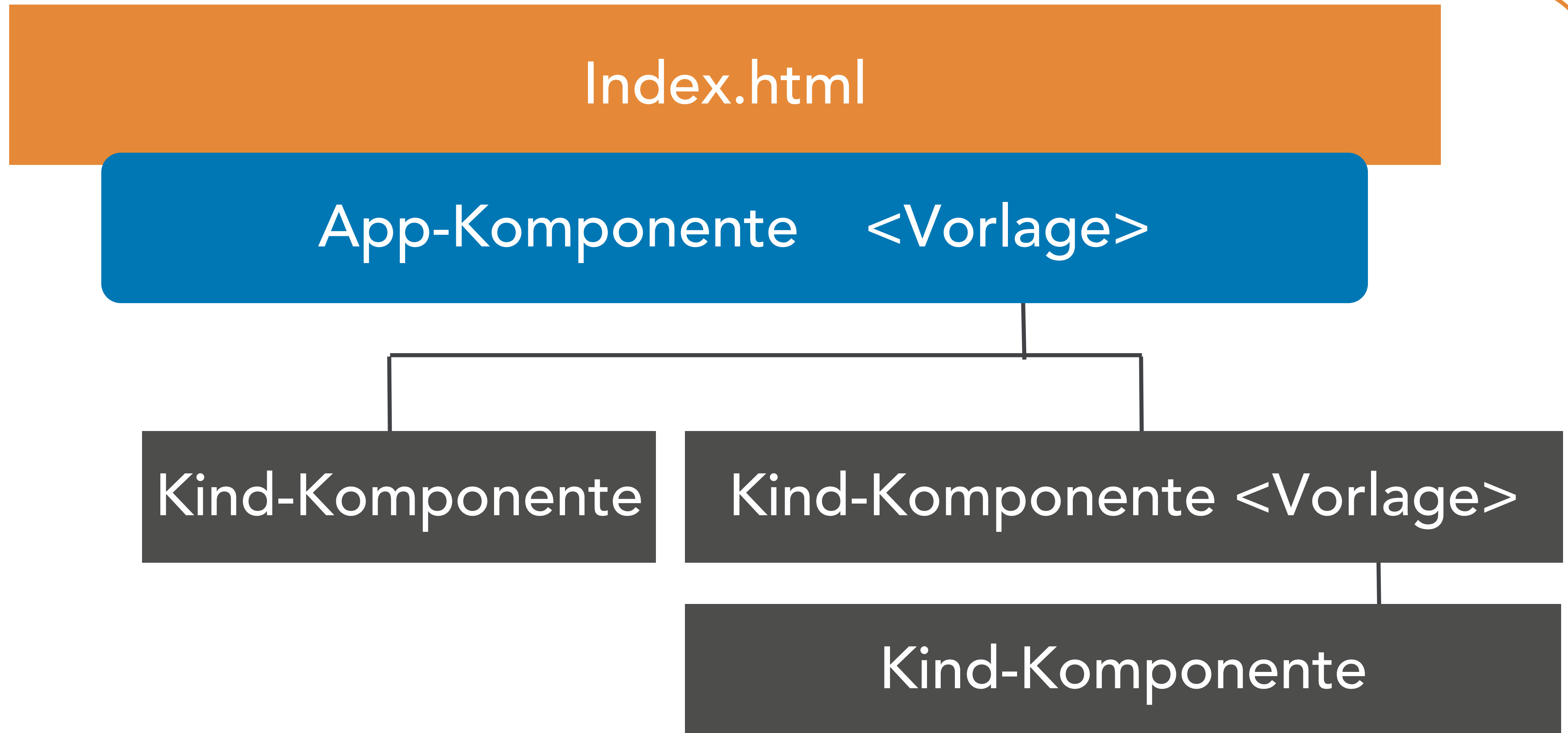
Komponentenbasierte Entwicklung

- 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/user --export --skipTests --flat`
 - selector
 - Knoten
 - Vorlage
 - templateUrl (file)
 - template (backticks)

Komponent Metadaten

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

Komponent Metadaten

- Style
 - encapsulation - Umgang mit Webkomponenten
 - ViewEncapsulation.Emulated
 - ViewEncapsulation.None
 - ~~• ViewEncapsulation.Native (deprecated)~~
 - ViewEncapsulation.ShadowDom

1

View

```
<user-list>
```

```
  <user-header></...>
```

```
  <user-item></...>
```

```
  <user-item></...>
```

```
  <user-item></...>
```

```
</user-list>
```

2

UserList - Template

```
<h3>user-list</h3>
```

```
<ng-content></...>
```

```
  <select="user-header">
```

```
    <user-item></...>
```

```
  </ng-content>
```

```
  <user-item></...>
```

```
</ng-content></ng-content>
```

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 da
- Mittels select Attribut lässt sich definieren, für welchen Inhalt der Platzhalter greifen soll

Zugriff auf transkludierte Inhalte

- Über Eigenschafts-Decorator
 - `@ContentChild`
 - Parameter: Komponentenkasse oder Selector
 - Options-Objekt
 - `static`: `false` | `true`
 - `read?`: `ElementRef` | `ViewContainerRef` | `Directive` | `Service`

Zugriff auf transkludierte Inhalte

- Über Eigenschafts-Dekorator
- Zugriff erst nach Hook
 - `ngAfterContentInit`
 - `ngOnInit` wenn `static true` ist
 - Aufgelöst vor dem Änderungserkennungslauf

Zugriff auf transkludierte Inhalte

- Über Eigenschafts-Dekorator
 - `@ContentChildren`
 - Parameter: Komponentenklasse
 - Optionales Options-Objekt mit Read-Eigenschaft
 - `read`: `ElementRef` | `ViewContainerRef` | `Directive` | `Service`
 - `descendants`: `false` | `true` (nur direkte Kinder === `false`)

Zugriff auf transkludierte Inhalte

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

Vorlagen Elemente ermitteln

- Über Eigenschafts-Decorator
 - @ViewChild
 - Parameter: Komponentenkasse / Hash-IDOptions-Objekt
 - static: false | true
 - read?: ElementRef | ViewContainerRef | Directive | Service

Vorlagen Elemente ermitteln

- Über Eigenschafts-Dekorator
- Zugriff erst nach Hook
 - `ngAfterViewInit`
 - `ngOnInit` wenn `static true` ist
 - Aufgelöst vor dem Änderungserkennungslauf

Vorlagen Elemente ermitteln

- Über Eigenschafts-Dekorator
 - @ViewChildren
 - Parameter: Komponentenkasse
 - 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>
```

lü' to the {{name}} placeholder, from 'this.name = 'Peter Müller;' to the chgName() method call, and from the chgName() method definition to the chgName() method call." data-bbox="340 380 800 580"/>

Bindungen

- Werte und Methode in Vorlagen binden
 - Mittels Ausdrucksinterpolation
 - `<h1>{{name}}</h1>`
 - `<h1>{{getName()}}</h1>`
 - ``

Bindungen

- Werte und Methode in Vorlagen binden
 - Als Eigenschaft binden
 - ``
 - Als Attribut binden
 - ``

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)=„METHODODE(\$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>
```


HostBindings- und Listener

- Mittels Eigenschaftsdekorator lassen sich auch Bindungen direkt in der Komponentenkasse 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 Asterisk (*) vor dem Attributnamen erkennbar:
 - ``
 - `<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 - ngIf

- [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, count

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 erstellen

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

Pipes erstellen

- Pipes sind **pure** d.h. wir haben eine Singleton und die Ausführung erfolgt bei Datenänderung.
- In den MetaDaten kann eingestellt werden das für pure false verwendet wird.
 - Somit ist die Pipe kein Singleton
 - Kann eigene Zustände somit handeln
 - Und wird durch die Änderungserkennung ausgelöst.

rxjs

<https://github.com/ReactiveX/rxjs>

<https://www.learnrxjs.io/>

<http://rxmarbles.com/>

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 - Observable

- 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 eines 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 TypeScript-Klassen
 - Instanzibereitstellung 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

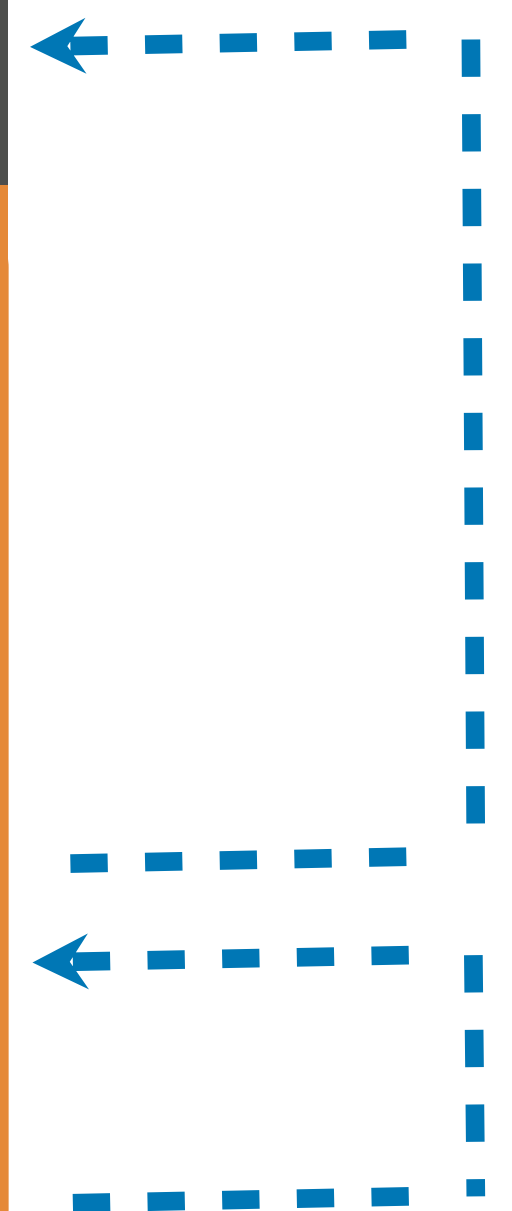
Rootinjektor [ServiceA]

KomponenteA-Injektor [ServiceA]
`@Component ({providers : [ServiceA]})`

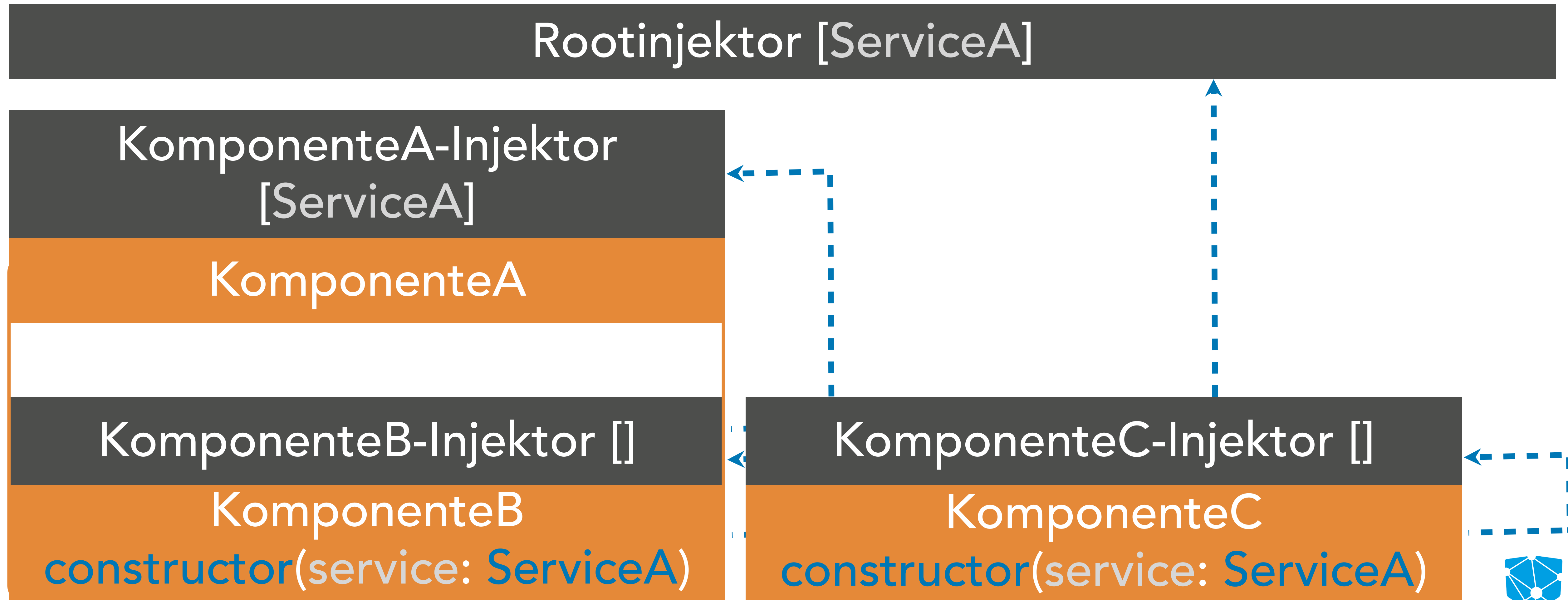
KomponenteA

KomponenteB-Injektor []

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



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. Def. Ziel-Injector
- **@Inject** - injiziert anhand eines Tokens
- **@Optional** – wird vor @Inject verwendet, ermöglichen optionale Injizierung
- **@Self, @Host, @SkipSelf** – wird genutzt, um das Injector-Bubling 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

Benutzen

- HttpClient-Modul importieren
- HttpClient-Service injizieren
- Methoden
 - Einen der CRUDServices nutzen
 - request<R>-Methode = Basis aller anderen Methoden
 - observable<R>

HttpRequest-Methoden

- Parameter `HttpRequest` oder:
 - `method`: string,
 - `'DELETE'|'GET'|'POST'|'PUT'`
 - `url`: string,
 - `options?`: Objekt zur detaillierten Spezifikation
- Rückgabe: `observable`

Request-Optionen

- body?: any;
- headers?: HttpHeaders;
- params?: HttpParams;
- reportProgress?: boolean
- withCredentials?: boolean

Request-Optionen

- responseType: 'arraybuffer' | 'blob' | 'json' | 'text';
- observe: 'body' | 'events' | 'response'
- Beide Parameter bestimmen Rückgabetyt für den Request

observe

responseType

return

body

arrayBuffer

Observable<ArrayBuffer>

body

blob

Observable<Blob>

body

text

Observable<string>

body

json

Observable<Object | R>

Request-Optionen

observe	responseType	return
events	arrayBuffer	Observable<HttpEvent<ArrayBuffer>>
events	blob	Observable<HttpEvent<Blob>>
events	text	Observable<HttpEvent<string>>
events	json	Observable<HttpEvent<Object R>>
response	arrayBuffer	Observable<HttpResponse<ArrayBuffer>>
response	blob	Observable<HttpResponse<Blob>>
response	text	Observable<HttpResponse<string>>
response	json	Observable<HttpResponse<Object R>>

Response-Typen

- `HttpResponse`
 - `body: T | null`
 - `headers: HttpHeaders`
 - `status: number`
 - `statusText: string`
- `url: string | null`
- `ok: boolean`
- `type: EventType.Response`

Response-Typen

- HttpEvent
 - Sent-Anfrage gesendet
 - UploadProgress – Upload-Fortschrittseignis (geladen#gesamt)
 - ResponseHeader – Antwortstatuscode und Header empfangen
 - DownloadProgress – Download-Fortschrittseignis (geladen#gesamt)
 - Response – Vollständige Antwort inkl. Body
 - User – Benutzerdefinierte Ereignisse

HTTP-Service Methoden

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

HttpInterceptor

- Anforderungen und Antworten lassen sich abfangen
- Service, dass das `HttpInterceptor` Interface implementiert
 - `intercept` - Methode
 - req: `HttpRequest<any>`,
 - next: `HttpHandler`
 - `-> Observable<HttpEvent<any>>`
 - `return next.handle(req);`

HttpInterceptor - bereitstellen

- 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({
 headers: {
 '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
);
 }
 })
);
}
```

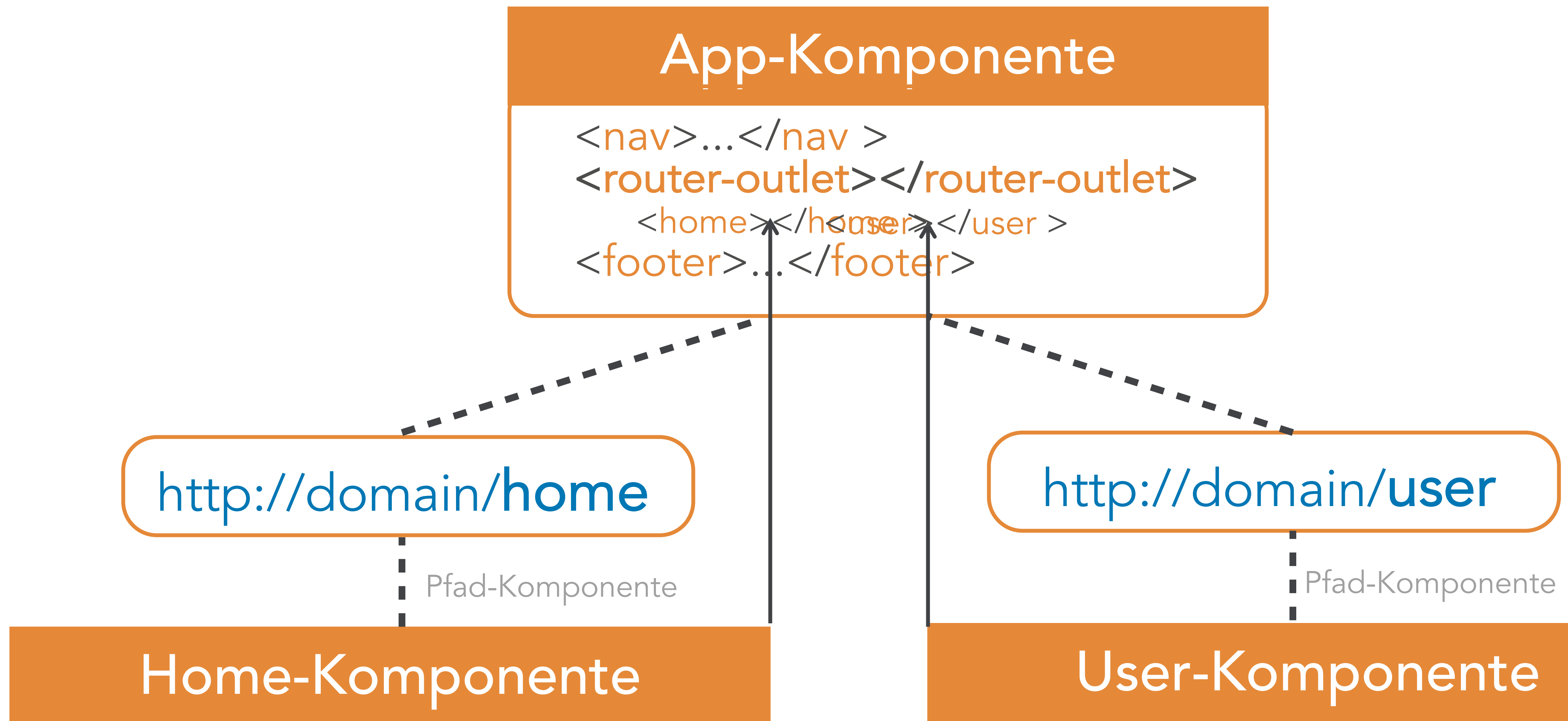
# **Routing**

## 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 inkl. Config
  - `Routes { path, component }`
    - `{ useHash: false }`
  - Optional sind Routen auch über den `Router`-Service und der `config` Liste zur Laufzeit konfigurierbar.
- `<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` : `import('./dash/dash.module')`  
                  `. then(m => m.DashModule)`
- Der Pfad zu dem Modul muss importiert und die Modul-Klasse als Promise zurückgegeben werden

# 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,
    - preloadingStrategy: 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') ) )`  
    `.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 Unterrouuten haben
- Diese müssen in der Config unter der Eigenschaft
  - children
    - analog zur vorhanden Konfiguration angelegt werden.

# Named Outlet

- Outlet mit namen versehen:  
`<router-outlet name="modal"></router-outlet>`
- Pfad auf outlet-Namen beziehen  
`{ path: 'modA', component: AComponent, outlet:'modal' }`
- Navigieren
- `[routerLink]="open</a>`
- `this.router.navigate( [ { outlets: { modal: null } } ] );`

# 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
 >

 <input name="email"
 #email="ngModel" ngModel>
 <input name="password"
 #password="ngModel" ngModel>

</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
 >

 <input name="email"
 #email="ngModel" ngModel>
 <input name="password"
 #password="ngModel" ngModel>

</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

# 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 widerspiegeln.
  - `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.



# 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

- Formular-Validierung wird über `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`
- Über den `ExistingProvider`
- Und `NG_VALIDATORS` Provide-Token:
- 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 gespeichert oder der View dargestellt 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;
  - Referenz muss aufgerufen werden, wenn der Benutzer im input-Feld Werte ändert.
  - Übermittelt wird der aktuelle Wert aus der UI.
  - Callback aktualisiert das Model über die Form-API

# Werte Zugriff – Interface Methoden

- `onTouched ()`: void;
  - Referenz muss aufgerufen werden, 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[msgFormatter]',  
 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"
 >

 <input type="email"
 formControlName="email"
 >
 <input type="password"
 formControlName="password"
 >

</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"
 >

 <input type="email"
 formControlName="email"
 >
 <input type="password"
 formControlName="password"
 >

</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
  - Drittes Element: AsyncValidator | AsyncValidator-Array

# Formulare – Helfer – 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; })`

**Veröffentlichen**  
JIT, AOT und mehr

# JIT

Server

Browser

Vorlagen  
Decorators  
Styles

Parse

View  
Code  
(AST)

Eval  
JS

View-  
Klassen

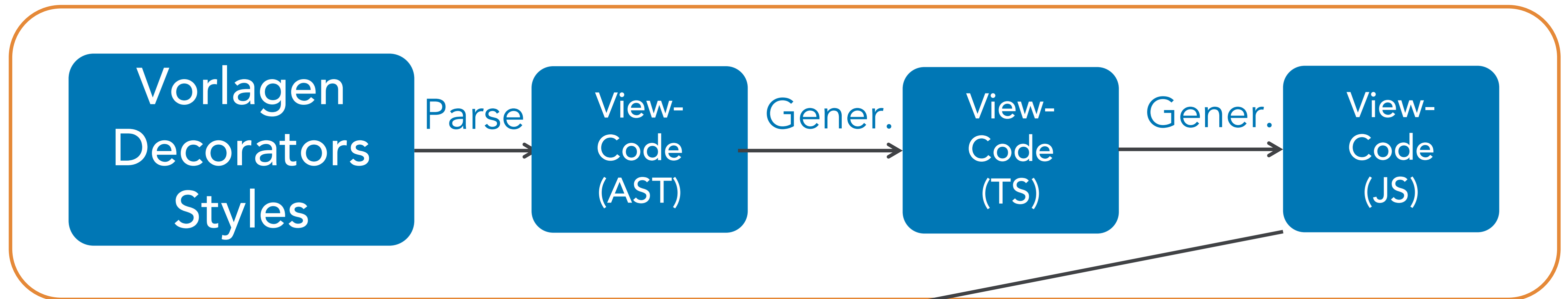
new

Laufende  
Anwendung

Kompilieren im Browser (Laufzeit)

# 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 $\geq$ 6)

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



# Veröffentlichen (Enviroment)

- 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`

# Debugging

# Debugging

- Möglichkeiten im Browser
- Browsererweiterungen nutzen

# **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 nativenElement)

# 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

# Dynamische Inhalte erstellen

- Vorlagen dynamisch nutzen
  - Strukturelle Direktive: `*ngTemplateOutlet`
    - Vorlagen-Referenz und Context
- `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()
    - markiert als dirty
    - Erkennung Eltern -> Kindern
  - detectChanges()
    - selbst -> Kinder

# 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

# DANKE

- <https://bit.ly/2Jzt12i>

