

Angular für Fortgeschrittene

Saban Ünlü

Zwei Worte zu mir

Saban Ünlü

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

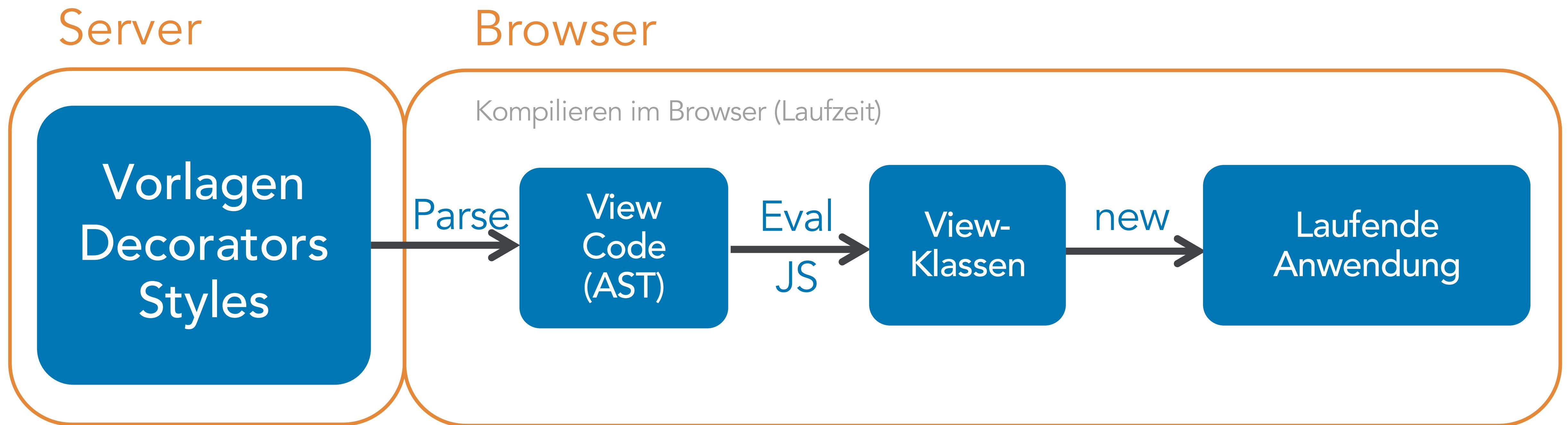
Quellen

- [unsere Beispiele] <https://github.com/netTrek/ng6-adv>
- [cheatsheet] <https://angular.io/guide/cheatsheet>
- [update guide] <https://update.angular.io/>
- [cli doku] <https://github.com/angular/angular-cli/wiki>
- [rxjs interagice diagramms] <http://rxmarbles.com/>
- [rxjs] <https://www.learnrxjs.io/>
- [form tutorial] <https://toddmotto.com/angular-2-forms-reactive>

Veröffentlichen

JIT, AOT und mehr

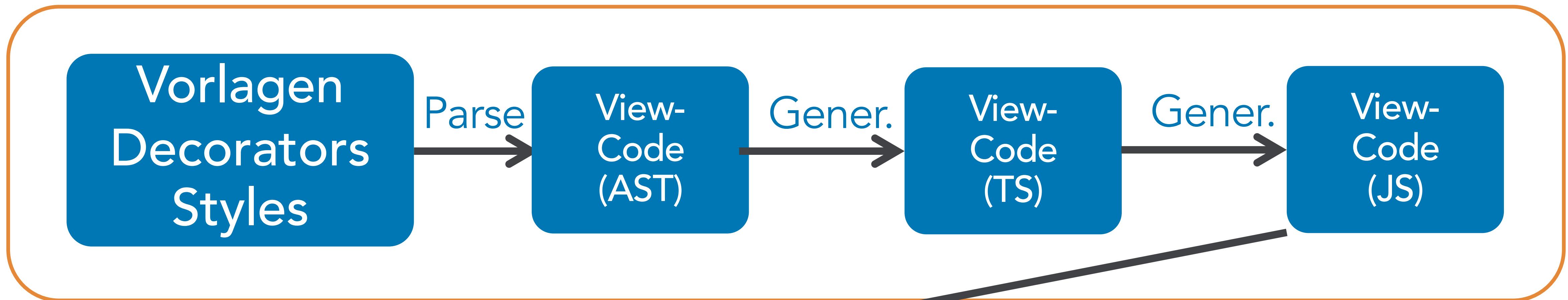
JIT



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

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-content select="user-header"></ng-content>  
  <user-item></...><br/>  
  </ng-content>  
<ng-content></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>

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`

Komponenteneignisse

- 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">

UserListVorlage

<user></user><user></user>

<user> </user>

</userList>



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 - Operatoren

- 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
 - ...

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()
 - 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

CRUD

Datenmodelle und der Datenfluss

HTTP-Service

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

HTTP-Service Methoden

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

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'`;

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
);
 }
 }
);
 };
}



netTrek



NewElements

Routing-Strategien

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 : UserDetailsService,
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]**

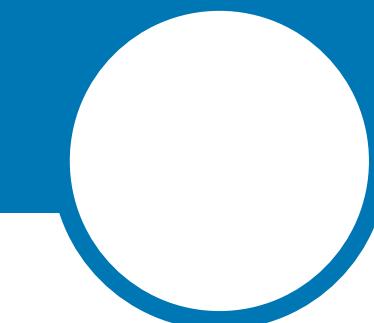


# Formulare - Vorlagen-getrieben

- 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.

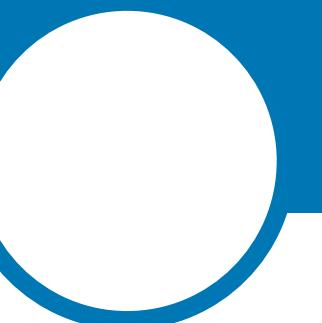


## Form

```
<form novalidate #myForm="ngForm">
 <input type="text"
 autocomplete="name"
 placeholder="name"
 name="name" ——————
 #name="ngModel"
 ngModel
 >

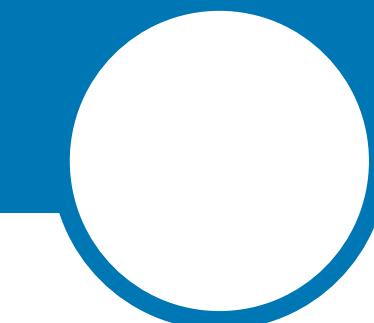
 <input name="email"
 #email="ngModel" ngModel>
 <input name="password"
 #password="ngModel" ngModel>

</form>
```



## Model

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

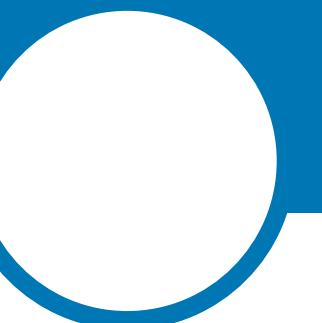


## Form

```
<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 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"
 >

 <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>
```

The diagram illustrates the data binding between an HTML form structure and a corresponding model object. On the left, a white rectangular area contains the HTML code for a form. On the right, a white rectangular area contains the resulting model object. A vertical blue bar separates the two. Five arrows point from specific elements in the form code to their corresponding properties in the model object. The first arrow points from the opening `<form>` tag to the `myForm.value` assignment. The second arrow points from the `<input type="text"` tag to the `name` property. The third arrow points from the `<span formGroupName="credentials">` tag to the `credentials` property. The fourth arrow points from the `<input type="email"` tag to the `email` property. The fifth arrow points from the `<input type="password"` tag to the `password` property.

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

# Model

# 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 – Helper – FormBuilder

- **Value:** Eigenschafts-Array
  - Erstes Element – Startwert
  - Zweites Element: Validator oder Validator-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; })`

# Testing

# Spec Dateien

- **describe** Block für einen Test erstellen
  - description: string
  - callback handler
- **describe callback** hat zwei 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` Helfer-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

- `import { async } from '@angular/core/testing';`
  - benötigt, um zu warten, dass compileComponents Template- und/oder Style-Dateien laden kann.
- `import { inject } from '@angular/core/testing';`
  - erforderlich, um Services zu injizieren
    - gib callback für it zurück

# 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 ();
 }));
}
```