Inhalt

[Konzept hinter Reactive Programming mit Observables: 3](#_Toc534840136)

[RxJS Operators 5](#_Toc534840137)

[Ajax Promises und Reactive Observables 5](#_Toc534840138)

[Eigene Observables 5](#_Toc534840139)

[Nach der Party wird aufgeräumt – unsubscribe 6](#_Toc534840140)

[Subject – das Allroundgenie 6](#_Toc534840141)

[Operatoren 7](#_Toc534840142)

[Angular 9](#_Toc534840143)

[Angular Projekt anlegen 9](#_Toc534840144)

[Module 10](#_Toc534840145)

[Aufbau eines Modules 10](#_Toc534840146)

[Components 11](#_Toc534840147)

[Aufbau einer Component 11](#_Toc534840148)

[Erstellen einer Component 15](#_Toc534840149)

[Kommunikation zwischen Components 15](#_Toc534840150)

[Styling 22](#_Toc534840151)

[Was passiert beim Start der Applikation? 23](#_Toc534840152)

[Syntax der TypeScript Klassen 24](#_Toc534840153)

[Bindings 25](#_Toc534840154)

[String Interpolation und Property Binding 25](#_Toc534840155)

[Event Binding 26](#_Toc534840156)

[Directives 28](#_Toc534840157)

[Bedingungen und Schleifen mit \*NgIf und \*NgFor 28](#_Toc534840158)

[Bedinungen mit ngSwitch 29](#_Toc534840159)

[Eigene Directives 30](#_Toc534840160)

[Services 34](#_Toc534840161)

[Beispiel 34](#_Toc534840162)

[Erstellen eines Services 35](#_Toc534840163)

[Bekanntmachen von Services 35](#_Toc534840164)

[Kommunikation von Components über Services 37](#_Toc534840165)

[Routing 38](#_Toc534840166)

[Verlinkung mit Benutzerinteraktion 38](#_Toc534840167)

[Verlinkung durch automatische Weiterleitung im Programmcode 39](#_Toc534840168)

[Reaktion auf Verlinkung im Template 40](#_Toc534840169)

[Übergabe von Parametern 40](#_Toc534840170)

[Formulare 51](#_Toc534840171)

[Template-Driven Ansatz 51](#_Toc534840172)

[Reactive Ansatz 56](#_Toc534840173)

[Umwandeln von Ausgaben mit Pipes 65](#_Toc534840174)

[Build-In Pipes von Angular – Standardfälle 65](#_Toc534840175)

[Internationalisierung 66](#_Toc534840176)

[Custom Pipes – Wenn’s komplizierter wird 67](#_Toc534840177)

[Parametrisierung von Custom Pipes 68](#_Toc534840178)

[Im anderen Kontext - Pipes ohne String Interpolation 69](#_Toc534840179)

[Asynchrone Pipes 70](#_Toc534840180)

[Der Schritt nach draußen – HTTP Requests in Angular 71](#_Toc534840181)

[Der klassische Ansatz (HttpModule) 71](#_Toc534840182)

[Die modernere Alternative (HttpClientModule) 74](#_Toc534840183)

[Debugging von TypeScript Code 78](#_Toc534840184)

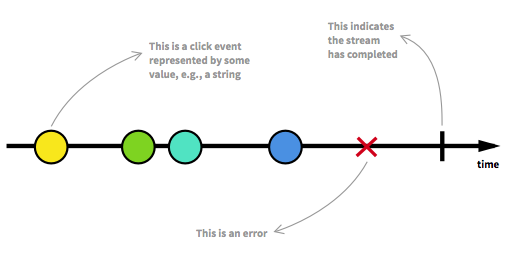
[ES6 (ECMAScript) Features 79](#_Toc534840185)

# Konzept hinter Reactive Programming mit Observables:

Hauptsächlich für die Entwicklung hoch-interaktiver Web Apps gedacht, in der eine große Menge von Echtzeitaktionen passieren und ausgewertet werden müssen z.B. direktes Speichern von Wertänderungen ohne separates manuelles Abschicken der Daten

Aktionen werden als Streams mit einem oder mehreren in zeitlicher Abfolge sortierten Aktionen behandelt z.B. alle Click Events. Es können drei verschiedene Signale auftreten:

1. Ein Wert eines bestimmten Typs wird ausgesendet
2. Ein Fehler wird ausgesendet
3. Der Stream ist zuende



--a---b-c---d---X---|->

a, b, c, d are emitted values

X is an error

| is the 'completed' signal

---> is the timeline

Es wird asynchron mit entsprechenden Funktionen auf jedes dieser Ereignisse reagiert.

Streams besitzen in den meisten Reactive Programming Implementierungen mehrere Funktionen dieser Art z.B:

* Map(c becomes 1): Ersetzt z.B. ein Zeichen durch ein anderes Zeichen. Im Grunde wird auf jedes Element x eines Streams A die Funktion f(x) ausgeführt und in einen Stream B überführt
* Filter: Filtert bestimmte Werte aus dem Stream heraus
* Scan(): Aggregiert alle Werte des Streams bis zu diesem Punkt (x = g(accumulated, current)). g() könnte die Werte einfach addieren

Diese Funktionen verändern den Stream an sich nicht, sondern geben einen neuen Stream mit den entsprechenden Veränderungen zurück z.B: clickStream.map(f).scan(g):

clickStream: ---c----c--c----c------c-->

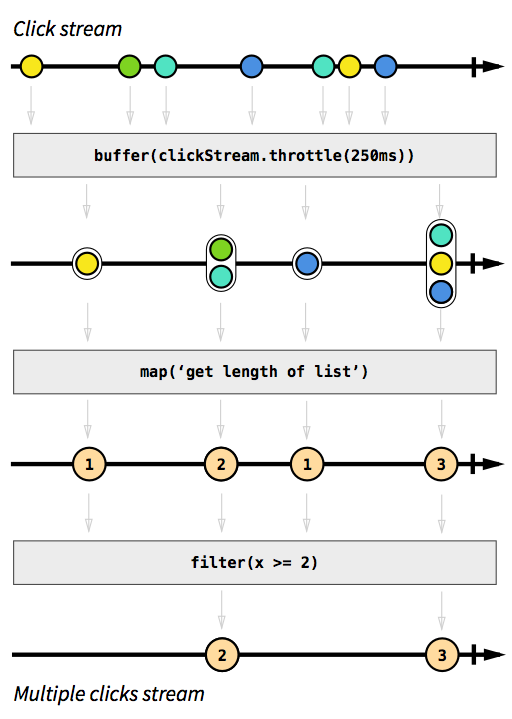
vvvvv map(c becomes 1) vvvv

---1----1--1----1------1-->

vvvvvvvvv scan(+) vvvvvvvvv

counterStream: ---1----2--3----4------5-->

Beispiel für eine Möglichkeit auf alle doppel oder dreifachklicks (und nur diese) reagieren zu können:



1. Liste an Click events werden gruppiert, wenn sie nicht weiter als 250 ms auseinanderliegen.
2. Anstelle der Gruppen selbst wird die Größe der Gruppen betrachtet
3. Es werden nur Gruppen mit mindestens 2 Klick Events betrachtet

## RxJS Operators

* switchMap(): Verbindet z.B. zwei Oberservables miteinander und “tauscht” deren Werte. Eine bestehende alte Subscription wird hierbei abgebrochen, d.h. wenn z.B. ein beliebiges Klick Event die Ausgabe eines Intervalls (mit z.B. 1 Sekunde zwischen den emittierten Werten) anstößt, dann würde ein zweiter Klick ein zweites Intervall starten. Mit switchMap wird das vorherige Intervall abgebrochen.

d--a---b-c---d---X---|->

a, b, c, d are emitted values

X is an error

| is the 'completed' signal

---> is the timeline

## Ajax Promises und Reactive Observables

Promises sind im Prinzip Observables, die jedoch nur einen Wert zurückgeben können. Ich denke, das ist vergleichbar mit Tasks in C# oder Futures in Java. Observables bzw. Streams können mehrere Werte zurückgeben.

Man kann ein Promise sehr leicht in ein Observable überführen:

Rx.Observable.fromPromise(promise)

## Eigene Observables

Es ist auch möglich, eigene Observables zu erstellen. Hierfür nutzt man die Observable.create() Methode. Jedes Oberservable verfügt über einen Observer, welcher die entsprechenden Werte, Fehlernachrichten oder den „Completed“ Status mitteilt.

const myObservable = Observable.create((observer: Observer<string>) => {

setTimeout(() => {

observer.next('first package'); // Pusht den nächsten Inhalt

}, 2000);

setTimeout(() => {

observer.next('second package'); // Pusht den nächsten Inhalt

}, 4000);

setTimeout(() => {

// Fail!

// observer.error('this does not work');

// Abschluss Signal 🡪 Observable ist fertig, d.h. das nächste Package kommt nicht an!!

observer.complete();

}, 5000);

setTimeout(() => {

observer.next('third pacakge'); // Pusht den nächsten Inhalt (kommt nicht an, da abgeschlossen bzw complete() aufgerufen wurde)

}, 6000);

});

myObservable.subscribe(

(data: string) => {console.log(data); } ,

(error) => {console.log(error); },

() => {console.log('Completed!'); }

);

### Nach der Party wird aufgeräumt – unsubscribe

Wenn Observables verwendet werden – vor allem wenn es sich um aufwändige oder unendlich laufende Aktionen handelt - ist darauf zu achten, dass beim Verlassen der Component die Subscription aufgehoben wird, da sie nicht automatisch beendet werden und das zu Speicherlecks führt. Dies kann erreicht werden, indem in der OnDestroy Methode die unsubscribe() Methode einer Subscription aufgerufen wird. Observables, die von Angular zur Verfügung gestellt werden, werden automatisch aufgeräumt, es sollte jedoch zur Gewohnheit werden, dies immer auch manuell zu tun.

numbersObservableSub: Subscription;

ngOnDestroy() {

this.numbersObservableSub.unsubscribe();

}

ngOnInit() {

const myNumbers = interval(1000);

this.numbersObservableSub = myNumbers.subscribe(

(number: number) => { console.log(number); },

(error) => {},

() => {}

);

…

}

### Subject – das Allroundgenie

Anstelle von Observables können auch Subjects zum Einsatz kommen. Ein Subject ist gleichzeitig Observable und Observer, d.h. es können sowohl Nachrichten empfangen, als auch gesendet werden. Sie eignen sich daher für Component-Übergreifende Kommunikation anstelle von Events, wenn man sie z.B. innerhalb eines Services platziert.

export class UsersService {

userActivated = new Subject();

}

users.service.ts

export class AppComponent implements OnInit {

user1Activated = false;

user2Activated = false;

constructor(private usersService: UsersService) {}

ngOnInit() {

this.usersService.userActivated.subscribe((id: number) => {

this.user1Activated = (id === 1);

this.user2Activated = (id === 2);

});

}

}

app.component.ts

export class UserComponent implements OnInit {

id: number;

constructor(private route: ActivatedRoute, private usersService: UsersService) { }

…

onActivate() {

this.usersService.userActivated.next(this.id);

}

}

user.component.ts

### Operatoren

RxJS verfügt über eine Vielzahl von Operatoren, welche die Ausgabe des Observables verändern. I.d.R. erzeugt jeder Operator ein neues Observable, somit können diese problemlos verkettet werden. In der zum Entstehungszeitpunkt dieses Dokuments aktuellen Version von Angular und RxJS werden die Operatoren mit der pipe()-Methode verkettet. Im folgenden Beispiel wird die Ausgabe das Interval-Observables mit dem Map-Operator so modifiziert, dass nur noch gerade Zahlen ausgegeben werden.

const myNumbers = interval(1000).pipe(map(

(data: number) => {

return data \* 2;

}

));

this.numbersObservableSub = myNumbers.subscribe(

(number: number) => { console.log(number); },

(error) => {},

() => {}

);

# Angular

Beschreibung einfügen…

## Angular Projekt anlegen

1. Node.js installieren
2. NPM/CLI installieren
3. “ng new projektname” in der Kommandozeile
4. „cd projektname“ in der Kommandozeile, um ein neues Angular Projekt zu erstellen
5. Optional („npm install --save bootstrap@3“ in der Kommandozeile um Bootstrap lokal für das Projekt zu installieren. Das erlaubt schönere Websites ohne groß CSS schreiben zu müssen)
6. In der Datei .angular-cli.json muss dann im Bereich „styles“ die bootstrap.css verlinkt werden. Hierbei auf ../ achten, da sonst im src Verzeichnis gesucht wird

**"styles"**: [  
 **"../node\_modules/bootstrap/dist/css/bootstrap.css"**,  
 **"styles.css"**],

## Module

### Aufbau eines Modules

Gruppiert eine oder mehrere Components. In kleinen Projekten genügt meist ein Module.

1. **Imports von TypeScript**: Damit ein Module von TypeScript erkannt wird, muss es zunächst importiert werden. Dies ist z.B. Voraussetzung dafür, dass sie innerhalb des NgModule-Decorator verwendet werden kann.

Beim Import in TypeScript wird die Dateiendung des zu importierenden Modules (.ts) weggelassen.

1. **@NgModule-Decorator** mit folgenden Eigenschaften
   1. **declarations**: Eine Component muss in einem Module registriert werden d.h. diesem zugehörig sein. Das geschieht innerhalb dieser Eigenschaft, anhand des Klassennamens innerhalb der entsprechenden Component TS.
   2. **imports**: Erlaubt das Importieren von anderen Modules. Das ist z.B. dann interessant, wenn wir eine größere Applikation haben, in der die Funktionalität übersichtshalber in mehrere Modules aufgeteilt haben. Es wird aber auch benötigt um bereits existierende Helfer-Module wie z.B. FormsModule (zur Nutzung der ngModel Directive) oder HttpModule (zur Versendung von http Anfragen) einbinden zu können.
   3. **providers**:
   4. **bootstrap**: Legt die Component fest, die beim Start der gesamten Applikation geladen werden soll, d.h. welche Component soll in der index.html zur Verfügung stehen.
2. Eine Module-Klasse. Diese wird z.B. bei den Imports eines anderen Moduls referenziert

## Components

Eine Applikation in Angular besteht aus mindestens einem Modul mit einem oder mehreren Components. Eine Component ist für die Darstellung der Applikation als Benutzeroberfläche verantwortlich.

### Aufbau einer Component

Eine Component besteht neben einer TypeScript Klasse auch aus je einer Template-HTML und CSS Datei. Die TypeScript Datei ist wie folgt aufgebaut:

1. **Imports von TypeScript**: Damit Klassen innerhalb von Components im TypeScript Code erkannt werden, müssen sie zunächst importiert werden.

Beim Import in TypeScript wird die Dateiendung der zu importierenden Klasse (.ts) weggelassen.

1. **@Component-Decorator** mit folgenden Eigenschaften
   1. **selector**: Definiert den (eindeutigen) Namen, mit welchem eine Component in einer Template HTML Datei eingebunden werden kann z.B. <app-root> in der index.html. Anstelle des selectors kann auch mit Routing gearbeitet werden

Selector arbeitet wie ein CSS Selector, d.h. es funktionieren auch andere Varianten davon:

|  |  |
| --- | --- |
| Selector in .ts Datei | In HTML |
| 'app-servers' | <app-servers></app-servers> |
| '[app-servers]' | <div app-servers></div> |
| '.app-servers' | <div class=''app-servers''></div> |

Id-Selektoren funktionieren NICHT!

* 1. **templateUrl oder template**: Legt die Template-HTML Datei (templateUrl) oder den Template HTML Inline Code (template) fest, der eingebunden wird, wenn eine Component in einer anderen Template HTML Datei (oder index.html) eingebunden wird. Bei der inline Variante wird keine separate HTML benötigt, sondern der Code wird direkt in die Component Datei (Dateiendung .ts) geschrieben. Wichtig hierbei ist, dass bei einfachen Hochkommata ( template: '<app-server></app-server><app-server></app-server>') keine Zeilenumbrüche erlaubt sind. Möchte man den HTML Code in mehrere Zeilen schreiben, so muss man mit Backticks arbeiten

template: `<app-server></app-server>

<app-server></app-server>`)

Die angegebene Template Datei ist dann mit der Component –Klasse „verheiratet“, d.h. sie kann auf Eigenschaften oder Event Handler der Component zugreifen (siehe z.B. Kapitel Bindings und Directives)

Wichtig: Es MUSS keinen Selector oder Style geben aber es MUSS ein Template geben

* 1. **styleUrls oder styles**: Style-Sheet zum Stylen der Template HTML Datei oder des inline Template HTML Codes. Wie bei templateUrl und template kann hier entweder eine Datei angegeben oder der Code direkt inline geschrieben werden.

#### Lifecycle einer Component

Component durchlaufen von ihrer Erstellung bis ihrem Ende verschiedene Phasen: der sog. Lifecycle. Angular besitzt sog. Lifecycle-Hooks, um Code zu definierten Zeiten (Phasen) ausführen zu können. Man muss die entsprechenden Interfaces im TypeScript Code explizit aufzulisten, wenn sie implementiert werden.

**export class** ServerElementComponent **implements** OnInit, OnChanges {

… }

|  |  |
| --- | --- |
| Hook | Feuert wenn … |
| ngOnChanges(changes: SimpleChanges) | … wenn sich eine gebundene Input-Variable (@Input) ändert. Im Parameter stehen Informationen wie z.B. der alte und der neue Wert. Wichtig ist, dass dies nur bei primitiven Datentypen funktioniert. Beim Binden an die Eigenschaft einer Klasseninstanz (Referenzverweis statt Wertverweis) ändert sich nur eine Eigenschaft der Instanz – nicht die Adresse der Instanz selbst. Daher wird in diesem Fall ngOnChanges NICHT ausgeführt. Das wäre nur der Fall, wenn die komplette Objektinstanz ausgetauscht würde. |
| ngOnInit() | … wenn die Component initialisiert wurde. Das bedeutet nicht, dass das Template bereits angezeigt wird bzw. in den DOM geladen wurde |
| ngDoCheck() | … Change Detection ausgeführt wird. Change Detection wird immer dann ausgeführt, wenn irgendeine Eigenschaft grafisch innerhalb des Templates anzeigt oder verändert wird z.B. immer wenn ein gebundener Wert an der Oberfläche aktualisiert wird oder wenn auf einen Button geklickt wurde oder wenn ein Timer abläuft usw… es wird auch ausgelöst, wenn sich der Wert nicht verändert hat,Hauptsache die View wurde aktualisiert |
| ngAfterContentInit() | … wenn der Content (ng-content) in die HTML Datei eingebettet wurde |
| ngAfterContentChecked() | … Change Detection für den eingebetteten Content ausgeführt wird d.h. wenn alle Änderungen für die Kind-View ausgeführt wurde oder nichts ausgeführt wurde, da es keine Änderung gab |
| ngAfterViewInit() | … wenn das Template der Component (und der Kind-Components) gerendert wurde |
| ngAfterViewChecked() | … Change Detection für die Component und die Kind-Components eingebetteten Content ausgeführt wird d.h. wenn alle Änderungen ausgeführt wurden oder nichts ausgeführt wurde, da es keine Änderung gab |
| ngOnDestroy() | … wenn die Component zerstört wird, da sie nicht mehr angezeigt wird bzw. wenn sie aus dem DOM entfernt wurde (z.B. wenn mit ng-if gearbeitet wurde und der Ausdruck wechselt auf false) |

### Erstellen einer Component

Eine Component kann manuell oder über das CLI erstellt werden. Die einfachere Methode ist die Nutzung des CLI. Hierzu öffnet man ein Kommandozeilenfenster und wechselt in das entsprechende Projektverzeichnis. Hier gibt man folgenden Befehl ein:

ng generate component [COMPONENT\_PFAD]/COMPONENT\_NAME [--spec=false]

Ein Pfad muss nicht angegeben warden, aber es hilft, die Applikation logisch zu unterteilen. Mit dem Parameter --spec wird angegeben, ob eine Testklasse erstellt werden soll. Im Falle von „false“ wird KEINE Testklasse erzeugt.

### Kommunikation zwischen Components

Da die einzelnen Funktionalitäten (Darstellen von Listen, Einfügen/Bearbeiten/Entfernen) innerhalb einer größeren Applikation aus organisatorischen Gründen in mehrere Components aufgeteilt werden, muss es eine Möglichkeit geben, dass diese Components Daten austauschen.

#### Daten von einer hierarchisch höher liegenden Component in eine eingebettete Component senden

Dies kann über Property Binding erreicht werden, indem das entsprechende Objekt beim Einbetten der Component übergeben bzw. an eine Property der Zielkomponente gebunden wird.

Wichtig ist hierbei, dass die Eigenschaft in der Zielkomponente mit dem Decorator @Input() versehen ist, da Properties standardmäßig nur innerhalb der Komponente verfügbar sind.

<**div class="container"**>  
  
 <**app-cockpit [serverElements]="serverElements"**></**app-cockpit**>  
 <**hr**>  
 <**div class="row"**>  
 <**div class="col-xs-12"** >  
 <**app-server-element \*ngFor="let *serverElement* of serverElements" [element]="*serverElement*"**></**app-server-element**>  
 </**div**>  
 </**div**>  
</**div**>

**export class** ServerElementComponent **implements** OnInit {  
  
 @Input() **element**: {**type**: **string**, **name**: **string**, **content**: **string**};  
  
 **constructor**() { }  
  
 ngOnInit() {  
 }  
  
}

<**div  
 class="panel panel-default"**>  
 <**div class="panel-heading"**>{{ **element**.name }}</**div**>  
 <**div class="panel-body"**>  
 <**p**>  
 <**strong \*ngIf="element.type === 'server'" style="color**: **red"**>{{ **element**.content }}</**strong**>  
 <**em \*ngIf="element.type === 'blueprint'"**>{{ **element**.content }}</**em**>  
 </**p**>  
 </**div**>  
</**div**>

**export class** AppComponent {  
 **serverElements** = [{**type**: **'server'**, **name**: **'Testserver'**, **content**: **'Just a test'**}];  
  
}

Man sieht, dass der @Input() Dekorator wie ein Methodenaufruf einer öffnenden und schließenden Klammer geschrieben wird. Innerhalb dieser Klammern kann ein Alias per String übergeben werden. In diesem Falle müsste die Eigenschaft über den Alias anstelle des eigentlichen Namens angesprochen werden, damit das Databinding funktionieren kann.

<**div class="container"**>  
  
 <**app-cockpit [serverElements]="serverElements"**></**app-cockpit**>  
 <**hr**>  
 <**div class="row"**>  
 <**div class="col-xs-12"** >  
 <**app-server-element \*ngFor="let *serverElement* of serverElements" [srvElement]="*serverElement*"**></**app-server-element**>  
 </**div**>  
 </**div**>  
</**div**>

**export class** ServerElementComponent **implements** OnInit {  
  
 @Input(**'srvElement'**) **element**: {**type**: **string**, **name**: **string**, **content**: **string**};  
  
 **constructor**() { }  
  
 ngOnInit() {  
 }  
  
}

#### Daten von einer hierarchisch tiefer liegenden Component in die „Vater“-Component senden (Custom Events)

Angenommen es gibt zwei Components C1 und C2, wobei C2 in C1 eingebettet ist. Findet nun eine Änderung in C2 statt, soll C1 darüber benachrichtigt werden. Dies kann durch Custom Events erreicht werden.

Dies kann mit Hilfe von EventEmittern geschehen. Diese werden in C2 mit dem @Output() Decorator deklariert und mit Hilfe der Methode emit() ausgelöst. Sie müssen via new-Operator als neues Event des Typs EvenEmitter erstellt werden, wobei der Typ definiert werden muss, welcher im $event Parameter übergeben wird. Der Methode emit() kann als Parameter die Daten mitgegeben werden, die in C1 benötigt werden. Hierbei muss es nicht zwingend eine ausgelagerte TypeScript Klasse handeln, sondern es kann auch ein einfaches Konstrukt sein.

**export class** CockpitComponent **implements** OnInit {  
  
 @Output() **serverCreated** = **new** EventEmitter<Server>();  
 @Output() **serverBlueprintCreated** = **new** EventEmitter<Server>();

// @Output() **serverBlueprintCreated** = **new** EventEmitter<Server>({type: **'**server**'**, serverName: **'**Testserver**',** serverContent: **'**Ein Testserver**'** });  
  
 onAddServer() {  
  
 **this**.**serverCreated**.emit(**new** Server(**'server'**, **this**.**newServerName**, **this**.**newServerContent**));  
  
 }

In C1 kann dann auf das Custom Event z.B. wie auf ein Click Event reagiert werden. In $event steht das Objekt, das vom EventEmitter übergeben wurde.

<**app-cockpit (serverCreated)="onServerElementAdded(*$event*)" (serverBlueprintCreated)="onServerElementAdded(*$event*)"**> </**app-cockpit**>

**export class** AppComponent {  
 **serverElements** = [**new** Server(**'server'**, **'Testserver'**, **'Just a test'**)];  
  
 onServerElementAdded(serverElement: Server) {  
 **this**.**serverElements**.push(serverElement);  
 }  
}

Wie bereits vom @Input() Decorator bekannt, kann innerhalb der geschweiften Klammern ein Alias übergeben werden, über dessen Namen das Event nach außen anstelle des eigentlichen Namens bekannt wird.

#### Lokale Referenzen innerhalb einer HTML Datei

Manchmal ist nicht zwingend ein Two-Way Binding nötig, um Daten innerhalb der gleichen HTML Datei zuzugreifen. Stattdessen können auch lokale Referenzen verwendet werden. Hierbei vergibt man einen eindeutigen Bezeichner, dem eine Raute vorgestellt wird (#MeineLokaleReferenz). Auf die Referenzen kann jedoch nur innerhalb der HTML Datei zugegriffen werden, NICHT im TypeScript Code. Es kann allerdings im Rahmen von Eventhandlern verwendet werden, um den Wert eines Felds als Parameter zu übergeben. Wichtig ist, dass lokale Referenzen für jedes HTML Element vergeben werden können. Das bedeutet allerdings auch, dass die lokale Referenz auf der HTML selbst verweist, nicht auf dessen Wert, d.h. auf das Eingabefeld anstelle des im Eingabefeld eingetragenen Werts. Daher muss der Wert separat vom Objekt ausgelesen werden. Innerhalb des HTML Dokuments kann man z.B. mit der doppelten geschweiften Klammer auf eine lokale Referenz verweisen.

<**input type="text" class="form-control" [(ngModel)]="newServerName" #serverNameInput**>  
  
  
<**label**>{{***serverNameInput***.value}}</**label**>

<**button  
 class="btn btn-primary"  
 (click)="onAddServer(serverNameInput.value)"**>Add Server</**button**>

onAddServer(nameInput: HTMLInputElement) {  
  
 **this**.**serverCreated**.emit(**new** Server(**'server'**,**nameInput**, **'Mein Content'**));  
  
}

Möchte man dennoch eine lokale Referenz im TypeScript Code ansprechen, muss diese mit Hilfe des @ViewChild() Decorators bekannt gemacht werden. Innerhalb der Klammern muss der Bezeichner der lokalen Referenz (ohne Raute) referenziert werden. Die zu erstellende Variable für die Referenz muss vom Typ „ElementRef“ sein.

Innerhalb des TypeScript Codes kann nun über die „nativeElement“-Eigenschaft auf den Wert des Felds zugegriffen werden.

**export class** CockpitComponent **implements** OnInit {  
  
 @Output() **serverCreated** = **new** EventEmitter<Server>();  
 @Output() **serverBlueprintCreated** = **new** EventEmitter<Server>();  
  
 @ViewChild(**'newServerContentInputField'**) **newServerContentInputField**: ElementRef;

onAddServer() {  
  
 **this**.**serverCreated**.emit(**new** Server(**'server'**, **this**.**newServerName**, **this**.**newServerContentInputField**.**nativeElement**.**value**));  
  
}

…

}

#### Komplexe Strukturen übergeben (Wiederverwendung)

Eine weitere Möglichkeit, Daten zwischen Components auszutauschen stellt die Anweisung ng-content dar. Hiermit lässt sich HTML in eine Component übergeben. Dafür schreibt man den HTML Code einfach zwischen die Einbindungsdirektive der Component und platziert innerhalb der empfangenden Component die Anweisung <ng-content> </ng-content> an die Stelle, an der der HTML Code stehen soll.

<**app-server-element \*ngFor="let *serverElement* of serverElements" [element]="*serverElement*"**>  
  
 <**p**>  
 <**strong \*ngIf="*serverElement*.serverType === 'server'" style="color**: **red"**>{{ ***serverElement***.**serverContent** }}</**strong**>  
 <**em \*ngIf="*serverElement*.serverType === 'blueprint'"**>{{ ***serverElement***.**serverContent** }}</**em**>  
 </**p**>  
  
  
</**app-server-element**>

app.component.html

<**div class="panel-body"**>  
  
 *<!-- Ausgelagert durch ng-content in app.component.html  
 <p>  
 <strong \*ngIf="element.serverType === 'server'" style="color: red">{{ element.serverContent }}</strong>  
 <em \*ngIf="element.serverType === 'blueprint'">{{ element.serverContent }}</em>  
 </p>  
 -->* <**ng-content**></**ng-content**>  
</**div**>

server-component.component.html

##### Auf Content zugreifen

Auch in der einbettenden Component kann der Content referenziert werden. Normalerweise würde man bei einer lokalen Referenz mit @ViewChild arbeiten. Da der Content allerdings in einer anderen Component eingebettet wird, gehört sie nicht direkt zur View des Kind-Component. Daher kann man mit @ContentChild auf das entsprechende Element auch innerhalb der einbettenden Kind-Component zugreifen.

<**app-server-element \*ngFor="let *serverElement* of serverElements" [element]="*serverElement*"**>  
  
 <**p #contentParagraph**>  
 …

</**p**>  
  
  
</**app-server-element**>

app.component.html

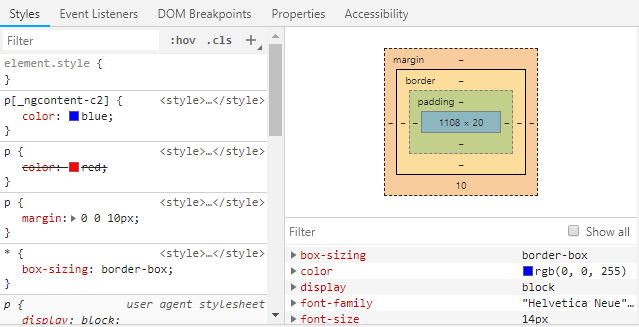
@ContentChild(**'content-paragraph'**) **paragraph**: ElementRef;

server-component.component.ts

### Styling

Alle Elemente einer Component werden über die entsprechende CSS Datei oder den inline-Style innerhalb der TypeScript Klasse der Component definiert. Das ist kein Standardverhalten des Browsers, sondern wird von Angular durch das automatische Hinzufügen von Selektoren realisiert. Möchte man stattdessen einen globalen Stil, so muss dieser in der styles.css Datei festgelegt werden. Diese Datei liegt direkt auf der Ebene des src Verzeichnisses. Wichtig ist hierbei, dass Styles, die tiefer in der Hierarchie erstellt werden den Vorzug vor den allgemeinen Styles erhalten.

Beispiel: Angenommen wir weisen jedem Paragraph Element der Applikation innerhalb der Styles.css die Schriftfarbe rot zu, aber in einer bestimmten Component (C2) überschreiben wir diesen Wert mit einer blauen Schriftfarbe innerhalb der in der Component festgelegten Styledatei oder dem inline Style. In diesem Fall wird die blaue Schriftfarbe (\_ngcontent-c2) übernommen anstelle des allgemeingültigen Styles.



Der technische Hintergrund für das beschrieben Verhalten ist die sog. View Encapsulation, in welcher für jede Componente ein einzigartiger Selektor definiert wird, was dafür sorgt, dass Styles, die in den entsprechenden Components erstellt werden, nicht für alle Elemente der Applikation gelten. Hierbei wird auf die Shadow DOM Technologie zurückgegriffen, die jedoch nicht von jedem Browser unterstützt wird. Angular kann sie jedoch simulieren. Man kann die View Encapsulation auch ausschalten. Dies kann innerhalb der TypeScript Datei einer Component innerhalb des @Component Decorators mit Hilfe des Attributs „encapsulation“ geschehen. Hierbei sind mit None, Emulated und Native drei Werte möglich.

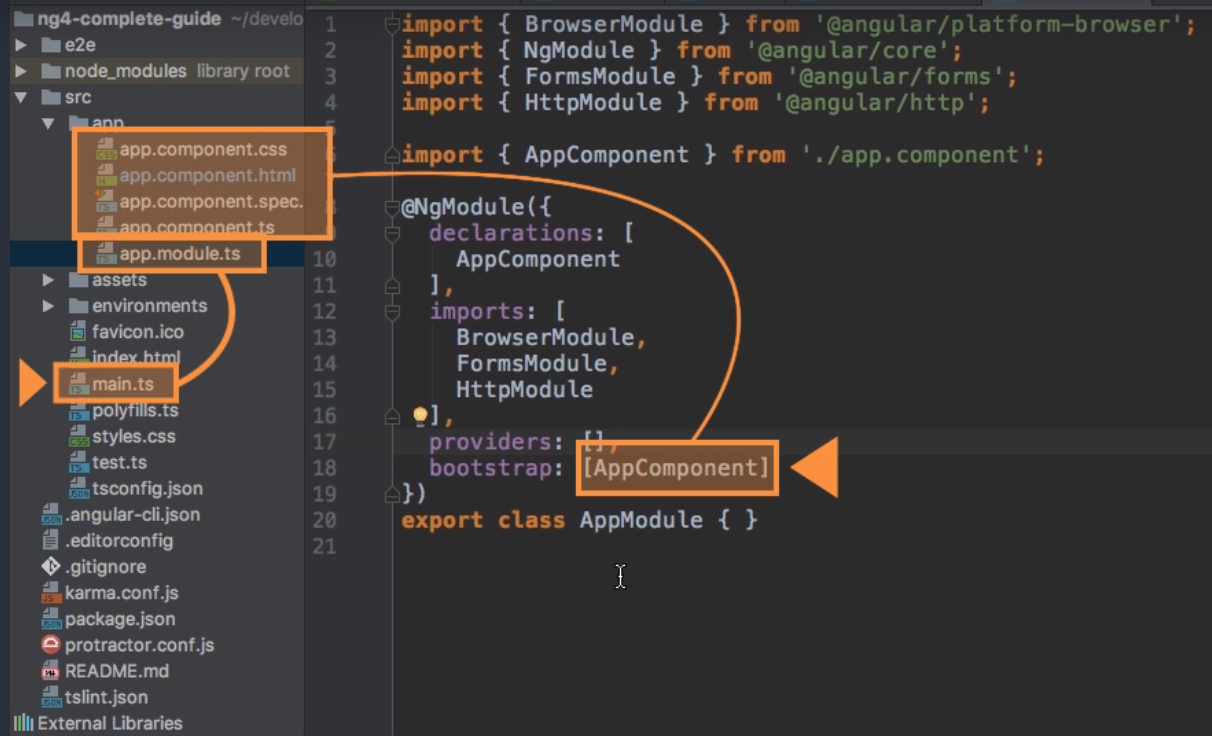
* Emulated: Zunächst beschriebenes Verhalten mit strikter Trennung der Style Dateien oder inline Styles zwischen den Components. Hierbei wird die Shadow DOM Technologie simuliert.
* Native: Gleiches Verhalten, allerdings wird die native Shadow DOM Implementierung des Browsers verwendet. Funktioniert nicht überall, da nicht jeder Browser Shadow DOM unterstützt.
* None: Schaltet View Encapsulation aus und macht somit den Style einer Component für die gesamte Applikation gültig.

@Component({  
 **selector**: **'app-server-element'**,  
 **templateUrl**: **'./server-element.component.html'**,  
 **styleUrls**: [**'./server-element.component.css'**],  
 **encapsulation**: ViewEncapsulation.*None*})

## Was passiert beim Start der Applikation?

Zunächst wird die main.ts aufgerufen, welche angibt, welches Modul geladen werden soll. Innerhalb des Moduls (app.module.ts) wird definiert, welche Komponente gestartet werden soll. Zu einem Modul gehört i.d.R. mindestens eine Komponente.

Dies erlaubt es, in der index.html auf die definierte Komponente zu verweisen, die dann inkludiert werden kann.



## Syntax der TypeScript Klassen

Beim Anlegen von Variablen ist es wichtig, dass nach einem Doppelpunkt (:) der Typ genannt und der Wert per Gleichheitszeichen (=) zugewiesen wird

export class MyComponent {

name = ‘Michael’;

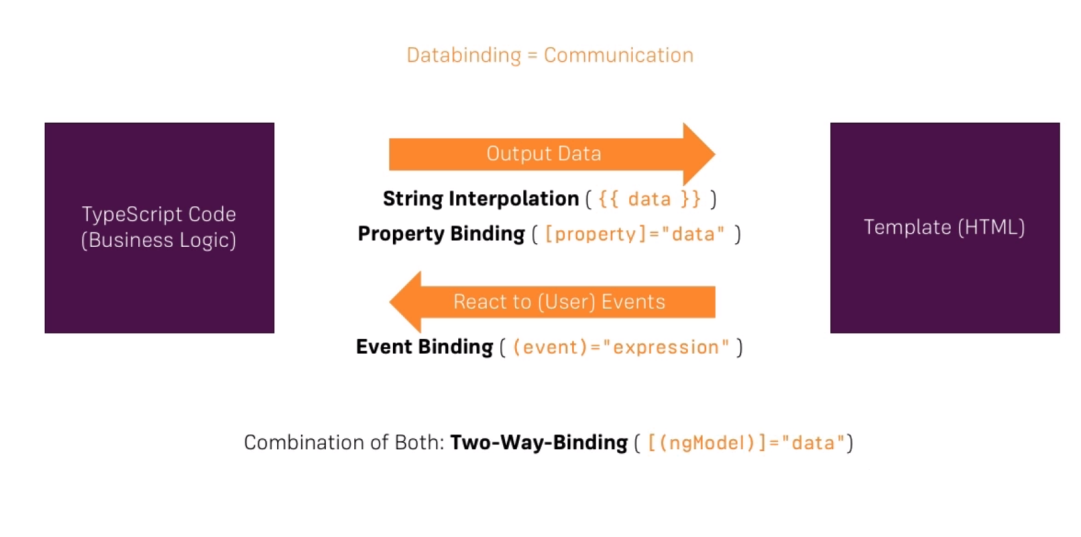
//or

myName: string = ‘Michael’;  
  
}

app.component.ts

## Bindings

Kommunikation zwischen Business Logic und Template



### String Interpolation und Property Binding

Bsp: Wir wollen, dass der Wert einer TextBox bei jeder Änderung in eine Variable geschrieben wird. Gleichzeitig soll ein Text Element (Paragraph in HTML) den Wert anzeigen.

export class AppComponent {

name = ‘Michael’;  
  
}

app.component.ts

<input type=”text” ([ngModel]) = “name”/>

<p> {{name}} </p>

app.component.html

Anstelle von StringInterpolations ( {{}}-Notation mit zwei geschweiften Klammern) kann auch mit sog. Property Bindings gearbeitet werden. Wenn z.B. ein Button disabled gesetzt werden soll, kann das über die „disabled“ Eigenschaft geschehen, die an ein boolean gebunden wird. Im Beispiel wird der Zustand nach 2 Sekunden auf True verändert. Es darf hier KEINE StringInterpolation verwendet werden!

**export class** ServersComponent **implements** OnInit {  
  
 **allowNewServer** = **false**;  
  
 **constructor**() {  
 *setTimeout*( () => {  
 **this**.**allowNewServer** = **true**;  
 }, 2000);  
 }  
}

app.component.ts

<**Button class="btn btn-primary" [disabled]="!allowNewServer"**>Add Server</**Button**>

app.component.html

### Event Binding

Möchte man auf Eingaben des Benutzers reagieren, kommt Event Binding zum Einsatz. Denkbar wäre das Reagieren auf Clicks auf einen Button oder die Reaktion auf die Änderung von Daten innerhalb eines Eingabefeldes. Die Syntax ähnelt der der Property Bindings. Beim Event Binding wird jedoch der Name des entsprechenden Events mit Klammern umgeben und dahinter der Code definiert, der ausgeführt werden soll, wenn das Ereignis eintritt. In den meisten Fällen wird hier eine Methode aufgerufen, es kann jedoch auch beliebiger TypeScript Code direkt im Template angegeben werden. Dies ist jedoch der Übersichtlichkeit abträglich und sollte daher - wenn überhaupt - nur sehr sparsam zum Einsatz kommen.

<**Button class="btn btn-primary"  
 (click)="onCreateServer()"**>Add Server</**Button**>

servers.component.html

onCreateServer() {  
 **this**.**serverCreationStatus** = **'Server was created'**;  
}

servers.component.ts

**Tip:**

The MDN (Mozilla Developer Network) offers nice lists of all properties and events of the element you're interested in. Googling for YOUR\_ELEMENT properties  or YOUR\_ELEMENT events  should yield nice results.

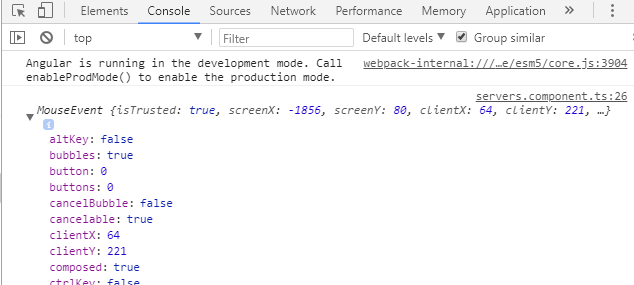
Beim Aufruf einer Funktion kann $event als Parameter mitgegeben warden. Dieses enthält Informationen über das jeweilige Ereignis z.B. die Koordinaten eines Clicks bei einem Click Event. Es ist sinnvoll, das event mit Hilfe von console.log(event) auszugeben, da somit leicht die Informationen innerhalb der Entwicklerkonsole eingesehen werden können.

<**Button class="btn btn-primary"  
 (click)="onCreateServer($event)"**>Add Server</**Button**>

servers.component.html

onCreateServer(event: any) {  
 **console.log(event);**  
}

servers.component.ts



Beispiel mit Text Input

Möchte man auf die Änderung von Text in einem HTML Text Input Feld reagieren, so muss ein Event Handler für das Event „input“ registriert werden. Am entsprechenden $event Parameter findet sich unter der Eigenschaft „target“ das Feld, welches das Event ausgelöst hatte. Möchte man nun den aktuellen Wert, so kann dieser über die „value“ Eigenschaft das targets abgeholt werden.

Ggf. muss explizit gecastet werden, damit target von der IDE akzeptiert wird

onCreateServer(event: any) {  
 **this**.**serverName** = event.**target**.**value**;

// ODER

**this**.**serverName** = (<HTMLInputElement>event.**target)**.**value**;

}

servers.component.ts

Eine weitere Möglichkeit ist 2-Way-Binding, wenn sich Daten ändern (durch Benutzereingabe in der View oder direkt in der Logik) wird die Änderung sowohl in der Oberfläche als auch in der Logik aktualisiert.

<**input type="text" class="form-control"  
 [(ngModel)]="serverName"**/>

servers.component.html

Hierbei bindet man direkt an eine Eigenschaft des Controllers. Wichtig ist hierbei, dass zur Verwendung von ngModel das FormsModule bekannt sein muss. Dies muss daher in dem Modul importiert werden, in welchem es zum Einsatz kommen soll.

## Directives

Directives sind Anweisungen die den DOM verändern. Das Einbinden von Components ist ebenfalls eine Directive, da der Inhalt in ein anderes HTML Dokument eingebettet wird.

### Bedingungen und Schleifen mit \*NgIf und \*NgFor

Ein mögliches Beispiel ist die Directive \*ngIf. Der Stern weist darauf hin, dass es sich hierbei um eine „structual directive“ handelt, d.h. sie verändert den DOM insofern, dass neue Elemente hinzugefügt werden, die vorher nicht enthalten waren.. \*ngIf erwartet ein Boolean. Wird z.B. ein Paragraph mit der Directive versehen, dann wird diese nur dann in den DOM eingefügt, wenn die Auswertung des Boolean „true“ ergibt. Tatsächlich ist das Element nicht im DOM verankert, solange die Bedingung nicht erfüllt ist. Ein weiteres Beispiel stellt \*ngFor da, welches eine Liste von Elementen durchläuft und entsprechend neue Elemente zum DOM hinzufügt.

Wichtig: Es kann immer nur eine strukturelle Directive zur gleichen Zeit an einem Element definiert werden, d.h. nicht gleichzeitig \*ngFor und \*ngIf.

<**p \*ngIf="serverCreated"**>Server was created, server name is {{**serverName**}}</**p**>

servers.component.html

onCreateServer(event: **any**) {  
 ***console***.log(event);  
 **this**.**serverCreationStatus** = **'Server was created" Name is '** + **this**.**serverName**;  
 **this**.**serverCreated** = **true**;  
}

servers.component.ts

Möchte man eine Else Anweisung umsetzen, geschieht dies (relativ umständlich) durch die Verwendung einer lokalen Sprungmarke.

<**p \*ngIf="serverCreated; else *noServer*"** >Server was created, server name is {{**serverName**}}</**p**>  
<**ng-template #noServer** >  
 <**p**>No server was created</**p**>  
</**ng-template**>

servers.component.html

Neben den “structural directives” gibt es auch “attribute directives”, die sich nur auf das Aussehen des Elements beziehen, auf welches sie angewandt wurden. Ein Beispiel hierfür stellt „ngStyle“ dar (ohne Stern).

<**input type=”text” [ngStyle]="{backgroundColor: ‘blue’ oder expression}" [ngClass]="{‘textColor’: expression}"** >

servers.component.html

.textColor {

color: blue;

}

servers.component.css

Intern wird die \*-Schreibweise eines If oder For Blocks umgewandelt, da es in Angular eigentlich keine \*-Schreibweise gibt. Stattdessen wird ein ng-template erstellt, und per PropertyBinding an dessen ngIf Eigenschaft gebunden.

<**ng-template [ngIf]="onlyOdd"**>  
 <**p**>Only odd</**p**>  
</**ng-template**>

app.component.html

### Bedinungen mit ngSwitch

Eine weitere Directive ist ngSwitch. Anders als \*ngIf und \*ngFor wird diese ohne \* an ein div angebracht. Dies liegt darin begründet, dass ngSwitch nur den Vergleichswert hält, ein oder ausgeblendet werden die Paragraphs mit ngSwitchCase. Bei ngSwitch handelt es sich demnach nicht um eine „structual directive“, bei ngSwitchCase und ngSwitchDefault jedoch schon.

<**div [ngSwitch]="value"**>  
 <**p \*ngSwitchCase="5"**>value is 5</**p**>  
 <**p \*ngSwitchCase="10"**>value is 10</**p**>  
 <**p \*ngSwitchDefault**>value is something else</**p**>  
</**div**>

app.component.html

### Eigene Directives

Es ist auch möglich, eigene Directives zu erstellen. Als Beispiel soll eine Directive entstehen, die die Hintergrundfarbe eines Elements beim Überfahren mit der Maus ändert.

#### Directives erstellen

Directives können wie Components manuell oder per CLI erzeugt werden. Der Befehl zum Erzeugen per CLI ist wie folgt:

ng generate directive [COMPONENT\_PFAD]/COMPONENT\_NAME [--spec=false]

#### Directives implementieren

Eine Directive ist eine eigene Klasse, welche mit dem Dekorator @Directive({…}) versehen wird. In dieser muss der Selector definiert werden, über welchen die Directive an den Elementen referenziert wird.

##### Direktzugriff auf Element per ElementRef

Über den Konstruktor steht der Directive das Element, auf welches sie angewandt wurde durch Injection automatisch zur Verfügung. Mittels private wird der Parameter des Konstruktors automatisch zu einer Klasseneigenschaft.

**import** {Directive, ElementRef, OnInit} **from '@angular/core'**;  
  
@Directive({  
 **selector**: **'[appBasicHighlight]'**})  
  
**export class** BasicHighlightDirective **implements** OnInit{  
 **constructor**(**private elRef**: ElementRef) {  
 }  
  
 ngOnInit(): **void** {  
 **this**.**elRef**.**nativeElement**.**style**.**backgroundColor** = **'green'**;  
 }  
}

basic-highlight.directive.ts

Directives besitzen Hooks für den Component Lifecycle, allerdings nur für OnInit und OnDestroy, da es kein Template gibt. Um die Directive anzuwenden, wird sie an ein Element als Attribut ohne Wert angehangen. Wichtig: Die Directive muss – wie eine Component – im Modul importiert werden, bevor sie benutzt werden kann.

<**p appBasicHighlight**>Style me with basic directive!</**p**>

app.component.html

##### Direktzugriff auf Element per Renderer

Es ist kein guter Programmierstil, die Elemente direkt über ihre Referenz zu verändern, da Angular auch im Bereich der ServiceWorker eingesetzt werden kann und nicht zwingend zu jedem Zeitpunkt der DOM direkt zur Verfügung steht. Stattdessen sollte ein Renderer zum Einsatz kommen, der kontextsensitiv arbeitet z.B. den DOM bearbeitet, sofern dieser verfügbar ist. Der Renderer kann ebenso wie die ElementRef über Injection bekannt gemacht werden.

**import** {Directive, ElementRef, OnInit, Renderer2} **from '@angular/core'**;  
  
@Directive({  
 **selector**: **'[appBetterHighlight]'**})  
**export class** BetterHighlightDirective **implements** OnInit{  
  
 **constructor**(**private elRef**: ElementRef, **private renderer**: Renderer2) { }  
  
 ngOnInit(): **void** {  
 **this**.**renderer**.setStyle(**this**.**elRef**.**nativeElement**, **'background-color'**, **'blue'**);  
 }  
}

better-highlight.directive.ts

<**p appBetterHighlight**>Style me with better directive!</**p**>

app.component.html

##### Auf Events des Elements reagieren mit @HostListener

Mit Hilfe des Dekorators @HostListener kann auf Events des Elements gelauscht werden, an welchem die Directive angebracht wurde. Hierbei kann es sich um jedes Event handeln, das am Element verfügbar ist z.B. click usw.

@HostListener(**'mouseenter'**) mouseover(eventData: Event) {  
 **this**.**renderer**.setStyle(**this**.**elRef**.**nativeElement**, **'background-color'**, **'blue'**);  
}  
  
@HostListener(**'mouseleave'**) mouseleave(eventData: Event) {  
 **this**.**renderer**.setStyle(**this**.**elRef**.**nativeElement**, **'background-color'**, **'transparent'**);  
}

better-highlight.directive.ts

##### An Eigenschaften des Elements binden mit @HostBinding

Mit dem Dekorator @HostBinding kann an beliebige Attribute des Elements gebunden werden, z.B. auch Styles.

@HostBinding(**'style.backgroundColor'**) **backgroundColor** = **'transparent'**;

…

**this**.**backgroundColor** = **'blue'**;

better-highlight.directive.ts

Je nachdem an was genau gebunden wird, müssen unterschiedliche Werte übergeben werden. Wenn man z.B. an alle Klassen (class) eines Elements binden möchte, handelt es sich um einen String.

@HostBinding(**'class'**) **classes** = **'class1 class2 class3'**;

example.directive.ts

Wenn man jedoch steuern möchte, ob eine Klasse gesetzt ist oder nicht, muss man an die Klasse selbst mit Hilfe eines Booleans binden.

@HostBinding(**'class.open'**) **isOpen** = **true**;

example.directive.ts

#### Kommunikation mit Directives

Mit Hilfe des @Input() Dekorators können Eigenschaftswerte in die Directive gesendet werden. Möchte man z.B. die Standard- und Hoverfarbe von außen setzen, kann dies folgendermaßen geschehen.

@Input() **defaultColor** = **'transparent'**;  
@Input() **highlightColor** = **'blue'**;

…

@HostListener(**'mouseenter'**) mouseover(eventData: Event) {  
**this**.**backgroundColor** = **this**.**highlightColor**;  
}  
  
@HostListener(**'mouseleave'**) mouseleave(eventData: Event) {  
**this**.**backgroundColor** = **this**.**defaultColor**;  
}

better-highlight.directive.ts

Im Beispiel werden durch die Zuweisung von „transparent“ und „blue“ Standardwerte hinterlegt, die dann greifen, wenn ein oder mehrere Parameter nicht übergeben wurden. Im HTML Code hat man die Möglichkeit, die Daten auf zwei Wegen zu übergeben. Nutzt man PropertyBinding mit den eckigen Klammern muss darauf geachtet werden, dass die Werte mit einzelnen Anführungsstrichen separat quotiert werden.

<**p appBetterHighlight [defaultColor]="'transparent'" [highlightColor]="'red'"**>Style me with better directive!</**p**>

app.component.html

Wenn sicher ist, dass definitiv statische Werte übergeben werden, kann man die eckigen Klammern und die Extra-Quotierung weglassen.

<**p appBetterHighlight defaultColor="transparent" highlightColor="red"**>Style me with better directive!</**p**>

app.component.html

Möchte man auf eine Eigenschaft (oder einen Alias) zugreifen die genau so heißt wie die Directive, muss der Name der Directive im HTML in eckige Klammern gesetzt werden.

@Input(‘appBetterHighlight’) **highlightColor** = **'blue'**;

better-highlight.directive.ts

<**p [appBetterHighlight]=’red’ defaultColor="transparent"** >Style me with better directive!</**p**>

app.component.html

#### Eigene strukturelle Directives entwickeln

Strukturelle Directives sind durch die \*-Schreibweise erkennbar (z.B. \*ngIf und \*ngFor). Es ist auch möglich, eigene strukturelle Directives zu erstellen z.B. eine Unless-Directive als Gegenteil zu \*ngIf. Wichtig ist hierbei, dass per Injection eine Referenz auf das entsprechende Template und auf einen ViewContainer (zur Ausgabe des Templates) übergeben wird. Der Inputparameter zur Übergabe der Bedingung wird mit dem Schlüsselwort set als Setter-Methode deklariert und muss demnach ausprogrammiert werden. Wichtig ist, dass dieser „Methodenname“ genau so heißt wie der Selector, damit die Directive erkannt wird.

**import** {Directive, Input, TemplateRef, ViewContainerRef} **from '@angular/core'**;  
  
@Directive({  
 **selector**: **'[appUnless]'**})  
**export class** UnlessDirective {  
  
 @Input() **set** appUnless(condition: **boolean**) {  
 **if** (!condition) {  
 **this**.**viewContainerRef**.createEmbeddedView(**this**.**templateRef**);  
 } **else** {  
 **this**.**viewContainerRef**.clear();  
 }  
 }  
 **constructor**(**private templateRef**: TemplateRef<**any**>, **private viewContainerRef**: ViewContainerRef ) {  
 }  
}

unless.directive.ts

## Services

Services sind durch Code Injection an beliebigen Stellen verwendbare Klassen mit einer bestimmten Aufgabe wie z.B. das Speichern von Daten in der Datenbank, Logging etc. Services sind normale TypeScript Klassen, die allerdings nicht selbst instanziiert werden. Die Erstellung und Bereitstellung Services wird vom Framework (in diesem Fall Angular) übernommen, wobei die Injektion über den Konstruktor geschieht.

### Beispiel

**export class** LoggingService {  
  
 log(message: **string**) {  
 ***console***.log(message);  
 }  
  
 **constructor**() { }  
}

logging.service.ts

Wichtig ist auch, dass der einbindenden Klasse der Service als Provider bekannt gemacht wird

@Component({  
 **selector**: **'app-new-account'**,  
 **templateUrl**: **'./new-account.component.html'**,  
 **styleUrls**: [**'./new-account.component.css'**],  
 **providers**: [LoggingService]  
})

…

**constructor**(**private loggingService**: LoggingService) { }

onCreateAccount(accountName: **string**, accountStatus: **string**) {  
 **this**.**accountAdded**.emit({  
 **name**: accountName,  
 **status**: accountStatus  
 });  
 **this**.**loggingService**.log(**'A server status changed, new status: '** + accountStatus);  
}

new-account.component.ts

### Erstellen eines Services

Ein Service kann manuell oder über das CLI erstellt werden. Die einfachere Methode ist die Nutzung des CLI. Hierzu öffnet man ein Kommandozeilenfenster und wechselt in das entsprechende Projektverzeichnis. Hier gibt man folgenden Befehl ein:

ng generate service [SERVICE\_PFAD]/SERVICE\_NAME [--spec=false]

Ein Pfad muss nicht angegeben werden, aber es hilft, die Applikation logisch zu unterteilen. Mit dem Parameter --spec wird angegeben, ob eine Testklasse erstellt werden soll. Im Falle von „false“ wird KEINE Testklasse erzeugt.

### Bekanntmachen von Services

Services können auf verschiedene Art und Weisen in der Applikation bekannt gemacht bzw. injiziert werden. Die Injektion funktioniert hierarchisch, d.h. wenn an einer Component ein Service durch Einfügen in das „providers“ Array bekannt gemacht wird, erhält diese UND ALLE Kind-Components die gleiche Instanz des Services. Wenn jedoch an einer Kind-Component der Service explizit injiziert wird, wird eine neue Instanz für die Kind-Component und deren Kinder-Components diese Instanz.

#### Direkter Import in einer Component als Provider

Gilt auch für alle Kind-Components aber NICHT für Services

@Component({  
 **selector**: **'app-new-account'**,  
 **templateUrl**: **'./new-account.component.html'**,  
 **styleUrls**: [**'./new-account.component.css'**],  
 **providers**: [LoggingService]  
})

new-account.component.ts

#### Import auf der obersten Hierarchie der Components (app.component.ts)

Gilt auch für alle Kind-Components aber NICHT Services

@Component({  
 **selector**: **'app-new-account'**,  
 **templateUrl**: **'./new-account.component.html'**,  
 **styleUrls**: [**'./new-account.component.css'**],  
 **providers**: [LoggingService]  
})

app.component.ts

#### Import auf der höchsten Hierachieebene (Modul)

Gilt für alle Components, Directives, Services,… des Moduls.

NgModule({  
 **declarations**: [  
 AppComponent,

…  
 ],  
 **imports**: [  
 BrowserModule,

…  
 ],  
 **providers**: [LoggingService],  
 **bootstrap**: [AppComponent]  
})

app.module.ts

Dies ist eine wichtige Voraussetzung dafür, dass Services in andere Services injiziert werden können! Hierzu ist es außerdem wichtig, dass in Services, in welchen andere Services injiziert werden sollen (und auch NUR dort) der Dekorator @Injectable() eingefügt wird.

**import** {LoggingService} **from './logging.service'**;  
**import** {Injectable} **from '@angular/core'**;  
  
@Injectable()  
  
  
**export class** AccountsService {  
  
 **constructor**(**private loggingService**: LoggingService) {  
  
 }

…

accounts.service.ts

**export class** LoggingService {  
  
 log(message: **string**) {  
 ***console***.log(message);  
 }  
  
 **constructor**() { }  
}

logging.service.ts

Alternativ kann mit Angular 6 auch über den @Injectable() Dekorator ein Service für das gesamte Modul zur Verfügung gestellt werden. Dies erlaubt auch Lazy Loading von Services, welches bei größeren Applikationen für die Performance von Bedeutung ist.

@Injectable({  
 **providedIn**: **'root'**})

logging.service.ts

### Kommunikation von Components über Services

Anstelle langer Ketten von Events, die von einem zur anderen Component umständlich weitergeleitet werden müssen, kann man mittels EventEmitter an einem Service die Kommunikation stark vereinfachen.

…

**constructor**(**private accountsService**: AccountsService) { }  
  
onSetTo(status: **string**) {  
 **this**.**accountsService**.updateStatus(**this**.**id**, status);  
 **this**.**accountsService**.**statusUpdated**.emit(status);  
}

account.component.ts

@Injectable()  
  
**export class** AccountsService {  
  
 **constructor**(**private loggingService**: LoggingService) { }  
  
 **statusUpdated** = **new** EventEmitter<**string**>();

accounts.service.ts

…

**export class** NewAccountComponent {  
 @Output() **accountAdded** = **new** EventEmitter<{**name**: **string**, **status**: **string**}>();  
  
 **constructor**(**private accountService**: AccountsService) {  
 **this**.**accountService**.**statusUpdated**.subscribe((status: **string**) => *alert*(**'New Status:'** + status));  
  
}

new-account.component.ts

## Routing

Angular ist ein Framework für Single-Page Applications, d.h. die komplette Applikation wird innerhalb einer einzigen HTML Seite abgehandelt, indem dynamisch Inhalte nachgeladen werden. Manche Teile der Single-Page Application sollen jedoch separat über URLs ansteuerbar sein. Hierfür existiert das Routing.

### Verlinkung mit Benutzerinteraktion

Routing wird innerhalb der app.module.ts bekannt gemacht. Zunächst müssen die Routen der Applikation definiert werden. Dies geschieht in einem Array des Types Routes. Hier muss der Pfad und eine Component angegeben werden, welche beim Ansteuern der Adresse geladen werden soll.

Innerhalb der imports des Modules werden die Routen mittels RouterModule.forRoot(…) bekannt gemacht. Ab diesem Zeitpunkt können sie verwendet werden.

**const** appRoutes: Routes = [  
 {**path**: **''**, **component**: HomeComponent},  
 {**path**: **'users'**, **component**: UsersComponent},  
 {**path**: **'servers'**, **component**: ServersComponent}  
];  
  
@NgModule({  
 **declarations**: [  
 AppComponent,  
 HomeComponent,  
 UsersComponent,  
 ServersComponent,  
 UserComponent,  
 EditServerComponent,  
 ServerComponent  
 ],  
 **imports**: [  
 BrowserModule,  
 FormsModule,  
 HttpModule,  
 RouterModule.*forRoot*(appRoutes)  
 ],  
 **providers**: [ServersService],  
 **bootstrap**: [AppComponent]  
})  
**export class** AppModule { }

app.module.ts

Der Ausgabepunkt (router-outlet) der zu ladenden Components muss ebenfalls angegeben werden. Eine sinnvolle Platzierung ist in der app.component.html

<**div class="row"**>  
 <**div class="col-xs-12 col-sm-10 col-md-8 col-sm-offset-1 col-md-offset-2"**>  
 <**router-outlet**></**router-outlet**>  
 </**div**>  
</**div**>

app.component.html

Möchte man nun einen der definierte Pfade ansteuern, kann das entweder über die Adresszeile des Browsers oder über via Interaktion mit der Website z.B. Klick auf einen Link geschehen. Wichtig ist, dass bei der Verlinkung eine Directive namens routerLink zum Einsatz kommt. Nutzt man z.B. ein normales <a href=“/users“>…</a> so wird die komplette Seite neu geladen. Dies ist nicht das gewünschte Verhalten. Die Directive kann auf verschiedene Arten verwendet werden.

<**ul class="nav nav-tabs"**>  
 <**li role="presentation" class="active"**><**a routerLink="/"**>Home</**a**></**li**>  
 <**li role="presentation"**><**a [routerLink]="'/servers'"**>Servers</**a**></**li**>  
 <**li role="presentation"**><**a [routerLink]="['/users']"**>Users</**a**></**li**>  
</**ul**>

app.component.html

* In der erste Variante wird der Pfad als Konstante übergeben also z.B. / oder /servers oder /users
* Die zweite Variante setzt auf Property Binding. Der Link wird hier zwar statisch angegeben, allerdings mit Hilfe von einfachen Anführungszeichen. Man könnte hier auch einen Ausdruck einfügen
* Die dritte Variante wird ebenfalls mit Property Binding gelöst. Diese Schreibweise sieht eher kompliziert aus, bringt aber Vorteile mit sich, wenn komplexere Pfade angesteuert werden sollen. Die Idee ist, dass ein Array mit den einzelnen Segmenten des Pfads übergeben wird. Ein tieferer Pfad würde auf folgende Weise realisiert: <a [routerLink]=“[ ‘/users‘,‘/details‘]“>.

Wichtig ist auch der Unterschied zwischen absolutem Pfad (/servers) und relativem Pfad (servers, also ohne Slash). Ein relativer Pfad bewegt sich immer relativ von der aktuellen Route, d.h. wenn man auf „/servers“ ist ergibt das Routing zum relativen Pfad „servers“ den Pfad „/servers/servers“, wohingegen das Routing zum absoluten Pfad „/servers“ (mit Slash) tatsächlich den Pfad „/servers“ ergibt. Im Falle des relativen Pfades wird also der angesteuerte Pfad an den aktuellen Pfad angehängt.

### Verlinkung durch automatische Weiterleitung im Programmcode

Alternativ zu manuellen Weiterleitung z.B. per Klick auf einen Link kann auch im Code eine Weiterleitung initiiert werden. Dies ist z.B. dann sinnvoll, wenn Daten gespeichert wurden und auf eine Übersichtsichtsseite weitergeleitet werden soll.

Hierzu muss im TypeScript-Code per Code Injection das Router Element injiziert werden. Per navigate() Funktion kann dann auf eine im Modul definierte Route weitergeleitet werden. Im Beispiel wird die bereits beschriebene Art und Weise verwendet, die Verlinkung per Array zu realisieren.

constructor(private router: Router) { }

onLoadServers() {

this.router.navigate(['/servers']);

}

home.component.ts

Vorsicht: Im Gegensatz zur Lösung mit routerLink, weiß der injizierte Router nicht, in welchem Pfad wir uns grade befinden, d.h. ein relatives Navigierien ist nicht möglich. Möchte man ein solches Verhalten ermöglichen, muss die Routeninformation ebenfalls injiziert werden. Um tatsächlich relativ zu navigieren, muss die navigate() Funktion um einen zweiten Parameter ergänzt werden, welcher angibt, zu welchem Pfad sich die ausgewählte Route relativ bewegen soll.

constructor(private serversService: ServersService,

private router: Router,

private route: ActivatedRoute) { }

onReload() {

this.router.navigate(['servers'], {relativeTo: this.route});

}

home.component.ts

### Reaktion auf Verlinkung im Template

Navigiert man innerhalb der Seite, so ist es sehr wahrscheinlich, dass dem Benutzer visuell signalisiert werden soll, an welcher Stelle er sich gerade befindet (z.B. Highlighting einer Kategorie). Dies kann z.B. mit der Directive „routerLinkActive“ geschehen. Mittels dieser wird einem Oberflächenelement ein gewisser Style (bzw. eine CSS Klasse) zugewiesen, wenn die aktuell angesteuerte Route dem festgelegten Pfad entspricht.

<li role="presentation" routerLinkActive="active" [routerLinkActiveOptions]="{exact: true}"><a routerLink="/">Home</a></li>

<li role="presentation" routerLinkActive="active"><a [routerLink]="'/servers'">Servers</a></li>

<li role="presentation" routerLinkActive="active"><a [routerLink]="['/users']">Users</a></li>

app.component.ts

Mittels der Directive „routerLinkActiveOptions“ kann dies näher definiert werden z.B. indem der Pfad genau identisch sein muss ({exact: true}). Wenn z.B. ein Pfad auf die Hauptseite mit einem einfachen Slash definiert wird („/“) dann enthält z.B. ein absoluter Pfad zur Übersicht der Server („/server“) auch den Slash. Das würde im oben stehenden Code OHNE routerLinkActiveOptions bedeuten, dass beide <li> – sowohl „Home“ als auch „Servers“ – die CSS Klasse zugewiesen bekommt. Gleiches würde auch für „/users“ oder sonstige Pfade und deren Verzweigungen („users/edit“) gelten.

### Übergabe von Parametern

Möchte man z.B. eine bestimmte Seite mit Informationen zu einem bestimmten Objekt ansteuern, so muss innerhalb des Pfads irgendwie übermittelt werden, um welches Objekt es sich im Detail handelt. Es ist möglich Parameter innerhalb von Routen zu übergeben und auszuwerten. Hierzu muss zunächst eine entsprechende Route mit dynamischen Bestandteilen im Modul erzeugt werden.

#### Verwendung von Pfadparametern

Pfadparameter erkennt man daran, dass sie wie ein Teil des eigentlichen Pfades aussehen d.h. es sind Zahlen oder Wörter, die mit einem Slash getrennt sind.

const appRoutes: Routes = [

{path: '', component: HomeComponent},

{path: 'users', component: UsersComponent},

{path: 'users/:id/:name', component: UserComponent},

{path: 'servers', component: ServersComponent}

];

app.component.ts

Im Beispiel ist z.B. nun folgende Route möglich: “/users/10/lena“.

Um die Parameterwerte auswerten zu können, müssen sie in der entsprechenden Component abgeholt werden. Dies funktioniert mittels Code Injection der gerade aktiven Route. Diese verfügt mittels die snapshot Eigenschaft über eine Abbildung des aktuellen Zustands. Es handelt sich tatsächlich nur um eine temporär gültige Abbildung zum Zeitpunkt des Ladens der Component. Eine Component wird nur dann neu geladen wenn sie von einem äußeren Pfad an navigiert wird. Platziert man also z.B. in der users Component einen Link, der einen bestimmten Benutzer laden soll, so wird die Component NICHT neu erstellt und die Parameterwerte entsprechen immer noch denen des ersten Ladevorgangs.

user: {id: number, name: string};

constructor(private route: ActivatedRoute) { }

ngOnInit() {

this.user = {

id: this.route.snapshot.params['id'],

name: this.route.snapshot.params['name']

};

}

users.component.ts

Soll auf Änderungen reagiert werden, so bedient man sich dem Observable „params“, welches direkt an der Route hängt (ohne „snapshot“). Mithilfe der subscribe() Funktion kann auf Veränderungen aktiv gelauscht werden.

<p>User with ID {{user.id}} loaded.</p>

<p>User name is {{user.name}}</p>

<hr />

<a routerLink="/users/7/lena">Load Lena</a>

users.component.ts

ngOnInit() {

this.user = {

id: this.route.snapshot.params['id'],

name: this.route.snapshot.params['name']

};

this.route.params.subscribe(

(params: Params) => {

this.user.id = params.id;

this.user.name = params.name; },

);

}

users.component.ts

Hierbei ist es sehr wichtig, darauf zu achten, dass es sich bei einem Parameterwert stets um einen String handelt. Dieser muss ggf. entsprechend vor der Übergabe noch in eine Number umgewandelt werden.

ngOnInit() {

const serverId = this.route.snapshot.params['id'];

this.server = this.serversService.getServer(Number.parseInt(serverId));

// OR: +this.serversService.getServer(serverId);

this.route.params.subscribe((params: Params) => {

this.server = this.serversService.getServer(Number.parseInt(params['id']));

});

}

server.component.ts

#### Verwendung von Query Parametern und Fragments

Neben den Pfadparametern gibt es auch sog. Query Parameter, die vom eigentlichen Pfad mittels eines Fragezeichens (?) getrennt sind, über einen Bezeichner verfügen und mit Hilfe von Kaufmanns-Und untereinander getrennt werden (z.B. „/servers/10/edit?allowEdit=true“). Um auf spezielle Anker im HTML Dokument zuzugreifen können auch sog. Fragments übergeben werden. Diese werden mittels einer Raute (#) an letzter Stelle des gesamten Pfads angehangen (z.B. „/servers/10/edit?allowEdit=true#loading“).

Auch auf diese Parameter kann im TypeScript Code zugegriffen werden. Zunächst kommen wir zur Übergabe der Parameter. Dies geschieht mittels der Eigenschaften „queryParams“ und „fragments“ an der „routingLink“ Directive.

<a

[routerLink]="['/servers', server.id, 'edit']"

[queryParams]="{allowEdit: '1'}"

[fragment]="'loading'"

href="#"

class="list-group-item"

\*ngFor="let server of servers">

{{ server.name }}

</a>

servers.component.ts

Dies ist auch möglich mittels TypeScript Code

constructor(private router: Router) { }

onLoadServer(id: number) {

this.router.navigate(['/servers', id, 'edit'],

{queryParams: {allowEdit: '1'},

fragment: 'loading'});

}

home.component.ts

#### Child-Routes

Routen können gruppiert werden, wenn sie voneinander abhängig sind d.h. alle „Unter“-Components einer Component „ServersComponent“ (z.B. Detailansicht, Bearbeitung, …) können als Kind-Routen definiert werden. Dies geschieht mit Hilfe des Attributs „children“ in der app.module.ts.

const appRoutes: Routes = [

{path: '', component: HomeComponent},

{path: 'users', component: UsersComponent},

{path: 'users/:id/:name', component: UserComponent},

{path: 'servers', component: ServersComponent, children: [

{path: ':id/edit', component: EditServerComponent},

{path: ':id', component: ServerComponent}

]}

];

app.module.ts

Gleichzeitig ist es nun jedoch erforderlich, dass die „Unter“-Components nun mit einem <router-outlet> in die Vater-Component eingebunden werden. Dies hat jedoch auch den Vorteil, dass die Components nun dynamisch in den Content geladen und aktualisiert werden können.

<div class="col-xs-12 col-sm-4">

<button class="btn btn-primary" (click)="onReload()">Reload</button>

<router-outlet></router-outlet>

<!-- VORHER:

<app-edit-server></app-edit-server>

<hr>

<app-server></app-server>-->

</div>

servers.component.html

#### Query Parameter bei Child Routes

Macht man in einer Kind-Component eine interne Weiterleitung per navigate(), gehen hierbei bereits gesetzte Query-Parameter, die beim Laden der darüberliegenden Component übergeben wurden verloren. Diese können mit der Option queryParamsHandling beibehalten werden, indem diese auf den Wert ‚preserve‘ gesetzt wird.

onEdit() {

this.router.navigate(['edit'], {relativeTo: this.route, queryParamsHandling: 'preserve'});

}

server.component.ts

Kommen weitere Parameter hinzu, kann der Wert ‚merge‘ verwendet werden.

#### Undefinierte Routen und Redirecting

Wenn ein Benutzer eine Route ansteuert die nicht existiert (undef. Route), ist es sinnvoll ihn zu einer Fehlerseite o.ä. weiterzuleiten (redirect).

Dies kann durch das Erstellen einer Wildcard Route (\*\*) geschehen. Hierfür definieren wir zwei neue Routen: Eine page-not-found Route und eine Wildcard Route für alle ungültigen eingegebenen URLS. Wichtig ist hierbei, dass die Wildcard Route als letzte Route definiert wird, da sonst JEDE Eingabe auf die Fehlerseite umgeleitet wird.

const appRoutes: Routes = [

{path: '', component: HomeComponent},

{path: 'users', component: UsersComponent, children: [

{path: ':id/:name', component: UserComponent}

]},

{path: 'servers', component: ServersComponent, children: [

{path: ':id/edit', component: EditServerComponent},

{path: ':id', component: ServerComponent}

]},

{path: 'not-found', component: PageNotFoundComponent},

{path: '\*\*', redirectTo: '/not-found'},

];

app.module.ts

**Wichtig:** Ein Redirect für den leeren Pfad (‘‘), also in unserem Beispiel anstelle der Route auf die HomeComponent sorgt per Default dafür, dass ALLE Routen weitergeleitet werden. Das liegt darin begründet, dass die Pfadauswertung einer URL automatisch ‚Prefix‘-gesteuert abläuft. JEDER Pfad beginnt mit einem leeren Zeichen (‘‘), daher wird jeder Pfad erkannt. Dies kann umgangen werden, indem zusätzlich das Attribut „pathMatch“ mit dem Wert „full“ verwendet wird. Mit diesem muss der Pfad komplett übereinstimmen bevor die Weiterleitung geschieht.

const appRoutes: Routes = [

{path: '', redirectTo: '/something, pathMatch: 'full'},

// anstelle von

// {path: '', redirectTo: '/something,},

app.module.ts

#### Auslagerung der Routen in ein separates Modul

Existieren viele Routen, ist der Übersichtlichkeit zuträglich, diese NICHT innerhalb der app.module.ts Datei zu verwalten. Stattdessen kann ein weiteres Modul z.B. „app-routing.module.ts“ erstellt werden.

Alle Routen aus „app.module.ts“ können nun nach „app-routing.module.ts“ kopiert werden. Nun ist es natürlich wichtig, das „RouterModule“ via „imports“ zu importieren und die Routen im neuen Module „forRoot“ zu registrieren, da diese in „app.module.ts“ nicht mehr vorhanden sind. Zusätzlich muss das (nun konfigurierte) „RouterModule“ nun auch via „exports“ nach außen verfügbar gemacht werden, damit „app.module.ts“ beim Importieren des neuen Modules darauf zugreifen kann.

const appRoutes: Routes = [

{path: '', component: HomeComponent},

{path: 'users', component: UsersComponent, children: [

{path: ':id/:name', component: UserComponent}

]},

{path: 'servers', component: ServersComponent, children: [

{path: ':id/edit', component: EditServerComponent},

{path: ':id', component: ServerComponent}

]},

{path: 'not-found', component: PageNotFoundComponent},

{path: '\*\*', redirectTo: '/not-found'},

];

@NgModule({

imports: [

RouterModule.forRoot(appRoutes)

],

exports: [

RouterModule

]

})

export class AppRoutingModule {

}

app-routing.module.ts

@NgModule({

declarations: [

AppComponent,

…

],

imports: [

BrowserModule,

…

AppRoutingModule

],

…

app.module.ts

#### Guards – Logik beim Ansteuern oder Verlassen von Routen

Angenommen, es gibt eine Applikation mit Benutzerverwaltung und einige Seiten dürfen nur dann angesteuert werden, wenn der Benutzer eingeloggt ist. Ein erster Ansatz wäre die Definition der Logik in der ngOnInit-Methode aller Ziel-Components. Dies sorgt allerdings für Redundanzen und schlecht wartbaren Code. Mit Route-Guards gibt es eine bessere Möglichkeit. Guards müssen als Service implementiert werden und können beim Ansteuern sowie beim Verlassen ausgelöst werden.

##### canActivate und canActivateChild

Diese treten ein, wenn eine Route (CanActivate) oder eine der Kindrouten (CanActivateChild) angesteuert werden. Im Beispiel befassen wir uns mit einem Mockup-Authentifizierungsservice namens AuthService, mit einfachen Methoden zum Ein- und ausloggen (login() und logout()) und einer Überprüfung ob ein Benutzer gerade eingeloggt ist (isAuthenticated()). Der Guard-Service „AuthGuard“ muss mindestens eines der beiden Interfaces CanActivate und CanActivateChild implementieren. Diese erfordern die Implementierung der entsprechenden gleichnamigen Methoden. Im Beispiel gibt die Methode isAuthenticated() asynchron ein Promise<boolean> zurück, welches dann ausgewertet wird. Wenn der Benutzer eingeloggt ist, darf die Route angesteuert werden. Falls nicht, wird auf die Startseite zurücknavigiert („/“).

@Injectable()

export class AuthGuard implements CanActivate, CanActivateChild {

constructor(private authService: AuthService, private router: Router) {}

canActivateChild(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {

return this.canActivate(route, state);

}

canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {

return this.authService.isAuthenticated()

.then((authenticated: boolean) => {

if (authenticated) {

return true;

} else {

this.router.navigate(['/']);

}

}

);

}}

auth-guard.service.ts

Damit die Route Guards funktionieren, müssen sie zusätzlich in der app-routing.module.ts für die Routen definiert werden. Dies funktioniert direkt an einer Vater-Route …

* … mit dem Attribut canActivate. In diesem Fall wird die komplette Route mit all ihren Kind-Routen gesperrt, sofern der Benutzer nicht angemeldet ist
* … mit dem Attribut canActivateChild. In diesem Fall werden die Kind-Routen gesperrt, sofern der Benutzer nicht angemeldet ist. Die Vater-Route kann weiterhin angesteuert werden.

const appRoutes: Routes = [

{path: '', component: HomeComponent},

{path: 'users', component: UsersComponent, children: [

{path: ':id/:name', component: UserComponent}

]},

{path: 'servers',

canActivate: [AuthGuard]

// ODER

canActivateChild: [AuthGuard],

component: ServersComponent, children: [

{path: ':id/edit', component: EditServerComponent},

{path: ':id', component: ServerComponent}

]},

{path: 'not-found', component: PageNotFoundComponent},

{path: '\*\*', redirectTo: '/not-found'},

];

app.module.ts

##### canDeactivate

Diese wird aktiviert, wenn von einer Route wegnavigiert werden soll. Dies kann insbesondere dafür genutzt werden, um den Benutzer davor zu bewahren, versehentlich von einer Seite wegzusteuern, wenn Änderungen noch nicht gespeichert wurden. Da in diesem Fall direkt auf Informationen der Component zugegriffen werden muss (z.B. wurden Daten geändert), muss die Kommunikation mit der Component gewährleistet werden. Dies wird erreicht, indem zusätzlich zu dem GuardService (CanDeactivateGuard) ein Interface (CanComponentDeactivate) erstellt wird, das die Component implementieren muss. Der Guard wird beim Verlassen der Route diese Methode aufrufen und somit ermitteln, ob weiternavigiert werden darf oder nicht. Dieses Pattern kann für beliebige Aufgabenstellungen verwendet oder angepasst werden.

export interface CanComponentDeactivate {

canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean;

}

export class CanDeactivateGuard implements CanDeactivate<CanComponentDeactivate> {

canDeactivate(component: CanComponentDeactivate,

currentRoute: ActivatedRouteSnapshot,

currentState: RouterStateSnapshot,

nextState?: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {

return component.canDeactivate();

}

}

can-deactivate-guard.service.ts

export class EditServerComponent implements OnInit, CanComponentDeactivate {

server: { id: number; name: string; status: string };

serverName = '';

…

constructor(

private serversService: ServersService,

private route: ActivatedRoute,

private router: Router,

private canDeactivateGuard: CanDeactivateGuard

) {}

…

canDeactivate(): Observable<boolean> | Promise<boolean> | boolean {

if (!this.allowEdit) {

return true;

}

if (

(this.serverName !== this.server.name ||

this.serverStatus !== this.server.status) &&

this.changesSaved === false

) {

return confirm('Do you want to discard the changes?');

} else {

return true;

}

}

edit-server.component.ts

In der app-routing.module.ts muss entsprechend der Guard für die Route bekannt gemacht werden.

const appRoutes: Routes = [

{path: '', component: HomeComponent},

…

{path: 'servers', canActivateChild: [AuthGuard], component: ServersComponent, children: [

{path: ':id/edit', component: EditServerComponent, canDeactivate: [CanDeactivateGuard]},

{path: ':id', component: ServerComponent}

app-routing.module.ts

##### Statische und dynamische Datenübergabe mit data und resolve

Neben der Parameter und Queryparameter können auch statische Daten an eine Component übergeben werden, die nicht Teil der URL sind. Hierfür muss in der app-routing.module.ts mit Hilfe des Parameters „data“ eine Key-value Liste übergeben werden, auf die in der Ziel-Component mit Hilfe des „data“-Attributs der aktiven Route (ActivatedRoute) zugegriffen werden kann.

{path: 'not-found', component: ErrorPageComponent, data: {message: 'Page not found!'}},

app-routing.module.ts

export class ErrorPageComponent implements OnInit {

errorMessage = '';

constructor(private activeRoute: ActivatedRoute) { }

ngOnInit() {

this.errorMessage = this.activeRoute.snapshot.data['message'];

this.activeRoute.data.subscribe((data) => {

this.errorMessage = data['message'];

});

}

}

error-page.component.ts

Es können alternativ auch dynamische Daten an eine Component übergeben werden. Dieser Vorgang kann als Alternative zum Laden von Daten innerhalb der ngOnInit()-Methode einer Component betrachtet werden. Mit Hilfe eines „Resolvers“ können Daten abgeholt werden, BEVOR die Component gerendert wird. Der hauptsächliche Unterschied zwischen „resolve“ und „canActivate“ liegt darin, dass es NICHT die Aufgabe eines Resolver ist, das Laden der Component abzubrechen bzw. im Problemfall auf eine andere Route umzuleiten. Er eignet sich besonders gut zum asynchronen Laden von Daten vor der eigentlichen Anzeige. Ein Resolver ist ein Service und implementiert das Interface “Resolve”. Hier muss ein Typ angegeben werden, welcher der Resolver zurückliefert. Im vorliegenden Fall wird der ServersService injiziert, um die Daten eines Servers für die Detail-Ansicht zurückzugeben. Wichtig ist hierbei, dass ein Resolver – so wie die Component auch – auf die Route und somit auch auf die Parameter zugreifen kann. Dies wird im vorliegenden Beispiel anhand des ID Parameters demonstriert.

interface Server {

id: number;

name: string;

status: string;

}

@Injectable()

export class ServerResolver implements Resolve<Server> {

constructor(private serversService: ServersService) { }

resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<Server> | Promise<Server> | Server {

return this.serversService.getServer(+route.params['id']);

}

}

server-resolver.service.ts

Das Abholen der Daten erfolgt – wie schon bei der statischen Variante – über das „data“-Attribut der aktivierten Route z.B. innerhalb der ngOnInit()-Methode der Component.

this.route.data.subscribe((data: Data) => {

this.server = data['server'];

});

server.component.ts

In der app-routing.module.ts muss der Resolver an der entsprechenden Route bekannt gemacht werden.

{path: 'servers', canActivateChild: [AuthGuard], component: ServersComponent, children: [

{path: ':id', component: ServerComponent, resolve: {server: ServerResolver}},

{path: ':id/edit', component: EditServerComponent, canDeactivate: [CanDeactivateGuard]},

]},

server.component.ts

## Formulare

Angular besitzt mächtige Werkzeuge zur Abhandlung von Formularen. Grundsätzlich existieren zwei Ansätze:

### Template-Driven Ansatz

Alle relevanten Controls in einem Formular werden manuell markiert und mit einem eindeutigen Namen versehen. Das geschieht zum einen mittels der Directive „ngModel“ und zum anderen durch die Eigenschaft „name“. Daraus wird automatisch ein Formular-Objekt erzeugt, welches beim Submit an die Component übergeben und dort ausgelesen wird. Das Formular selbst wird per Anker bekannt gemacht. Der Name „ngForm“ verweist auf das JSON Objekt des Formulars.

#### Formulardaten

<form (ngSubmit)="onSubmit(f)" #f="ngForm">

<div id="user-data">

<div class="form-group">

<label for="username">Username</label>

<input type="text" id="username" class="form-control" ngModel name="username">

</div>

…

app.component.html

export class AppComponent {

onSubmit(form: NgForm) {

console.log(form);

}

}

app.component.ts

Eine Alternative hierzu stellt die Verwendung von @ViewChild dar, bei welcher das Formular nicht bei onSubmit() übergeben wird, sondern direkt als Element zugreifbar gemacht wird.

export class AppComponent {

@ViewChild(‘f’) signupForm: NgForm;

onSubmit() {

console.log(this.signupForm);

}

}

app.component.ts

#### Validatoren

Angular bietet einige eingebaute Validatoren für Formulare, z.B. um die korrekte Syntax von E-Mail Adressen und/oder das Ausfüllen der Datenfelder zu forcieren. Verstöße gegen diese Validierungsmaßnahmen spiegeln sich im Status des Formulars anhand der Flags „valid“ und „invalid“ direkt wieder. Alle eingebauten Validatoren sind in der Validators Klasse einsehbar ( <https://angular.io/api/forms/Validators>). Es ist auch möglich, die Standardvalidatoren von HTML5 zu aktivieren, indem die Directive „ngNativeValidate“ zu einem Control hinzugefügt wird.

<div class="form-group">

<label for="email">Mail</label>

<input type="email"

id="email"

class="form-control"

ngModel

name="email"

required

email>

app.component.html

Wenn z.B. eine Warnung über den aktuellen Status des Formulars angezeigt werden soll, also z.B. „Bitte geben Sie eine gültige E-Mail Adresse ein“, so kann dies mit Hilfe der „\*ngIf“ Direktive geschehen. Dem entsprechenden Control wird ein Anker hinzugefügt, welcher auf das ngModel des entsprechenden Controls verweist.

<div class="form-group">

<label for="email">Mail</label>

<input type="email"

id="email"

class="form-control"

ngModel

name="email"

required

email

#email="ngModel">

>

<span class="help-block" \*ngIf="!email.valid && email.touched">Please enter a valid email!</span>

app.component.html

Die Eigenschaft “touched” gibt an, ob ein Control bereits angeklickt wurde, d.h die Nachricht wird erst dann ausgegeben, wenn das Eingabefeld bereits besucht wurde. Hierfür muss nicht zwingend Text eingegeben worden sein, ein Klick genügt bereits.

#### Vorselektion

Mittels One-Way-Binding können Standardwerte innerhalb eines Formulars gesetzt werden. Es gilt die Richtung TS -> Oberfläche, d.h. manuelle Eingaben in der Oberfläche haben keine Auswirkung auf die Variable im TypeScript Code. Hierbei wird ngModel per Property Binding mit der Variable verknüpft :

<input type="email"

id="email"

class="form-control"

[ngModel]="defaultMail"

name="email"

required

email

#email="ngModel">

app.component.html

export class AppComponent {

@ViewChild(‘f’) signupForm: NgForm;

defaultMail = ‘account@provider.de’;

}

app.component.ts

Vorsicht: Der Text in der Oberfläche wird bei jeder Änderung der Variable defaultMail synchronisiert, auch wenn der Benutzer zuvor anderen Text eingegeben hat. Dies trifft allerdings nur dann zu, wenn sich der neue Wert von defaultMail tatsächlich vom vorherigen Wert unterscheidet, d.h. wenn in der Variable bereits ‘account@provider.de‘ steht, hat eine erneute Zuweisung des gleichen Wertes keine Auswirkung auf die Oberfläche.

#### Selektives Setzen von Werten

Es ist sowohl möglich das gesamte Formular als auch einzelne Teile daraus aus JS auszufüllen.

<form (ngSubmit)="onSubmit(f)" #f="ngForm">

<button class="btn btn-default" (click)="suggestUserName" type="button"> Suggest an username </button>

…

app.component.html

export class AppComponent {

@ViewChild(‘f’) signupForm: NgForm;

suggestUserName() {

const suggestedName = 'Superuser';

// setValue() setzt alle Werte

this.myForm.setValue( { username: this.suggestUserName, email:

'test@test.de', secret: 'pet' });

// form.patchValue() setzt einzelne Werte (hier username)

this.myForm.form.patchValue( { username: this.suggestUserName });

}

}

app.component.ts

#### Dauerhafte und beidseitige Synchronisierung mittels Two-Way-Binding

Mittels Two-Way-Binding kann die Oberfläche und das Model in TypeScript beidseitig synchronisiert werden d.h. Eingaben in der Oberfläche wirken sich aufs Model aus und Änderung im Model spiegeln sich direkt in der Oberfläche wieder. Hierbei kommt eine Kombination von Property Binding (eckige Klammern) und Eventhandling (normale Klammern) zum Einsatz.

<input type="email"

id="email"

class="form-control"

[(ngModel)]="defaultMail"

name="email"

required

email

#email="ngModel">

app.component.html

export class AppComponent {

@ViewChild(‘f’) signupForm: NgForm;

defaultMail = ‘account@provider.de’;

}

app.component.ts

#### Handling von Radiobuttons

TODO

#### Formulare zurücksetzen

Der Status eines Formulars (mit all seinen Werten ) kann mittels der reset() Methode des Formular-Objekts zurückgesetzt werden.

<form #f="ngForm">

…

<button class="btn btn-warning" (click)="f.reset()">Reset</button>

</form>

app.component.html

### Reactive Ansatz

Im reaktiven Ansatz wird das Formular mittels entsprechender repräsentativer Felder im Model verknüpft, wobei auch die Hierarchie beachtet werden muss. Um reactive Forms verwenden zu können, muss das entsprechende Modul importiert werden.

#### Formulardaten, Vorselektion und Validatoren

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

ReactiveFormsModule,

HttpModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

app.module.ts

<div class="container">

<div class="row">

<div class="col-xs-12 col-sm-10 col-md-8 col-sm-offset-1 col-md-offset-2">

<form [formGroup]="myForm" (ngSubmit)="submitted()">

<div class="form-group">

<label for="username">Username</label>

<input

type="text"

id="username"

formControlName="username"

class="form-control">

</div>

<div class="form-group">

<label for="email">email</label>

<input

type="text"

id="email"

formControlName="email"

class="form-control">

</div>

<div class="radio" \*ngFor="let gender of genders">

<label>

<input

type="radio"

formControlName="gender"

[value]="gender">{{ gender }}

</label>

</div>

<button class="btn btn-primary" [disabled]="!myForm.valid"

type="submit">Submit</button>

</form>

</div>

</div>

</div>

app.component.html

import { Component, OnInit } from '@angular/core';

import { FormGroup, FormControl } from '@angular/forms';

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent implements OnInit {

genders = ['male', 'female'];

myForm: FormGroup;

ngOnInit(): void {

this.myForm = new FormGroup({

'username': new FormControl(null, Validators.required),

'email': new FormControl(null, [Validators.required, Validators.email]),

'gender': new FormControl('male')

});

}

submitted() {

console.log(this.myForm.value);

}

}

app.component.ts

Im Beispiel wird außerdem das Handling von **Validatoren** im reactive Forms Umfeld gezeigt. Der erste Parameter des FormControl Konstruktors legt den Default-Wert fest, während der zweite Parameter die Validatoren bestimmt. Im Grunde stehen über die Klasse Validators alle Validatoren zur Verfügung, die auch im Template Driven Ansatz verwendet werden können.

#### Einzelne Controls abfragen

Zum Abfragen einzelner Controls existiert die get() Methode in der FormGroup – Klasse. Hiermit kann über den Namen das jeweilige Control abgeholt werden um z.B. auf den Validierungstatus oder den Wert zugreifen zu können.

…

<label for="username">Username</label>

<input

type="text"

id="username"

formControlName="username"

class="form-control">

</div>

<p class="help-block" \*ngIf="!myForm.get('username').valid && myForm.get('username').touched">

Please enter a valid username!

</p>

…

app.component.html

Im Beispiel wird überprüft, ob ein Benutzername eingegeben wurde, wobei dem Benutzer die Chance gegeben wird, einen Wert einzutragen, bevor die Nachricht erscheint.

#### Controls gruppieren

Die erwähnte get() Methode kann auch für tiefere Verschachtelungen von Controls verwendet werden. In diesem Fall wird zusätzlich zum Namen auch der Pfad zum Element angegeben.

…

<div formGroupName="userData">

<label for="username">

Username

<input

type="text"

id="username"

formControlName="username"

class="form-control">

</label>

<p class="help-block" \*ngIf="!myForm.get('userData.username').valid && myForm.get('userData.username').touched">

Please enter a valid username!

</p>

<label for="email">

Email

<input

type="text"

id="email"

formControlName="email"

class="form-control"

>

</label>

</div>

…

app.component.html

ngOnInit(): void {

this.myForm = new FormGroup({

'userData': new FormGroup({

'username': new FormControl(null, Validators.required),

'email': new FormControl(null, [Validators.required, Validators.email]),

}),

'gender': new FormControl('male')

});

console.log(this.myForm);

}

app.component.ts

#### Dynamisches Hinzufügen von Controls

Manchmal ist es nötig, zur Laufzeit weitere Controls zu einem Formular hinzuzufügen. Dies ist mit dem reaktiven Ansatz kein großes Problem. In der Hierarchie können Gebiete als Array definiert werden, in welchen neue Elemente hinzugefügt werden können.

<div formArrayName="hobbies">

<h4>Your Hobbies</h4>

<button class="btn btn-default" type="button" (click)="onAddHobby()">Add Hobby</button>

<div class="form-group" \*ngFor="let hobby of

myForm.get('hobbies').controls; let i = index">

<input type="text" class="form-control" [formControlName]="i" />

</div>

</div>

app.component.html

In der HTML Datei wird zunächst ein Feld definiert, in welchem die Controls erscheinen sollen. Mit der Eigenschaft „formArrayName“ wird das Feld mit dem TypeScript Code synchronisiert. Im TypeScript Code gibt es aktuell noch kein Element mit dem Namen „hobbies“, dieses wird im zweiten Codelisting erzeugt. In dem Feld befindet sich außerdem ein Button, um ein neues Control anzulegen.

Mittels einer Schleife wird auf das noch zu erstellende Array „hobbies“ zugegriffen und für jeden Eintrag ein Control erzeugt. Außerdem wird eine Zählvariable angelegt, welche als Name des Controls („formControlName“) gelten soll. Normalerweise würde dieser Wert statisch im TypeScript Code stehen, was jedoch logischerweise im dynamischen Kontext nicht möglich ist.

ngOnInit(): void {

this.myForm = new FormGroup({

'userData': new FormGroup({

'username': new FormControl(null, Validators.required),

'email': new FormControl(null, [Validators.required, Validators.email]),

}),

'gender': new FormControl('male'),

'hobbies': new FormArray([])

});

}

onAddHobby() {

const control = new FormControl(null, Validators.required);

(<FormArray>this.myForm.get('hobbies')).push(control);

}

app.component.ts

Im TypeScript Code wird in der FormGroup ein neues Element „hobbies“ erstellt, welches vom Typ „FormArray“ ist. Dieses wird mit einem leeren Array initialisiert. Wird die Methode onAddHobby() ausgeführt, so wird ein neues FormControl angelegt. Um es nun zu dem Array hinzufügen zu können, muss es zunächst mittels der get() Methode abgeholt und auf „FormArray“ gecastet werden. Dies ist eine der wenigen Situationen, in denen ein expliziter Cast notwendig ist.

**ACHTUNG:**

Es ist möglich, dass in manchen Fällen (z.B. wenn es sich im Array nicht um einfache Controls, sondern um formGroups handelt) die Schleife durch alle Controls des Arrays fehlschlägt. In diesem Fall muss das Abholen der Controls in eine eigene Methode ausgelagert werden. Dies liegt daran, dass Angular innerhalb der Schleife TypeScript nicht versteht.

<div class="form-group" \*ngFor="let hobby of

getHobbyControls(); let i = index">

<input type="text" class="form-control" [formControlName]="i" />

</div>

app.component.html

getHobbyControls() {

return (<FormArray>this.myForm.get('hobbies')).controls;

}

app.component.ts

#### Benutzerdefinierte Validatoren

Genügen die vorgegeben Validatoren nicht, können eigene erstellt werden. Diese können als zweiter (synchrone Validatoren) oder dritter (asynchrone Validatoren z.B. mit Abfrage an einen Webservice) Parameter im Konstruktor eines FormControl mitgegeben werden.

1. Synchrone Validatoren

Hierbei wird der entsprechende Validator als Funktion angelegt, welcher das Control als Parameter erwartet. Als Rückgabewert der Funktion wird ein JavaScript Objekt mit einer Key-Value Beziehung definiert die aus einem String als Schlüssel und einem boolean als Wert besteht. Im Falle, dass der Validator nicht zum Zuge kommt, wird null zurückgegeben. Wichtig ist, dass bei der Übergabe der Funktion der Kontext mit bind(this) gebunden wird, wenn in der Validatorfunktion auf Klassenvariablen zugegriffen werden soll. Dies ist nötig, da Angular die Methode in einem anderen Kontext aufruft. Eine weitere Möglichkeit wäre die Definition der Funktion als anonyme Funktion mit Fat-Arrow Operator (=>), da hier der Kontext bestehen bleibt.

this.myForm = new FormGroup({

'userData': new FormGroup({

'username': new FormControl(null, [Validators.required,

this.forbiddenNames.bind(this)

]),

'email': new FormControl(null, [Validators.required, Validators.email]),

}),

…

forbiddenNames(control: FormControl) : {[s: string]: boolean} {

if(this.forbiddenUsernames.indexOf(control.value) !== -1) {

return {'nameIsForbidden': true};

}

return null;

}

app.component.ts

1. Asynchrone Validatoren (mit Promises; dritter Parameter bei FormControl)

Diese verhalten sich ähnlich zu den synchronen Validatoren, es wird jedoch anstelle eines JavaScript Objekts ein Promise oder Observable als Rückgabewert definiert.

'email': new FormControl(null, [Validators.required, Validators.email], this.forbiddenEMailsAsyncValidator.bind(this)),

…

forbiddenEMailsAsyncValidator(control: FormControl): Promise<any> | Observable<any> {

const promise = new Promise((resolve, reject) => {

setTimeout(() => {

if (this.forbiddenEMailAddresses.indexOf(control.value) !== -1) {

resolve({'emailAddressIsForbidden': true});

}

resolve(null);

}, 1500);

});

return promise;

}

app.component.ts

Möchte man auf diese selbst definierten Errorcodes zugreifen kann dies einfach über die errors Eigenschaft am entsprechenden Control geschehen.

<p class="help-block" \*ngIf="!myForm.get('userData.email').valid && myForm.get('userData.email').touched">

<span \*ngIf="myForm.get('userData.email').errors['required']">

E-Mail is required!

</span>

<span \*ngIf="myForm.get('userData.email').errors['email']">

Please enter a valid email address!

</span>

<span \*ngIf="myForm.get('userData.email').errors['emailAddressIsForbidden']">

Choosen email address is forbidden!

</span>

</p>

app.component.html

#### Reaktion auf Wert und Statusänderungen

Es existieren zwei Observables (valueChanged und statusChanged) auf die reagiert werden kann.

this.myForm.valueChanges.subscribe(() => console.log(this.myForm));

this.myForm.statusChanges.subscribe(() => console.log(this.myForm.status));

#### Selektives Setzen von Werten

setValue (gesamtes Formular), patchValue (selektives Setzen von Werten), reset (Zurücksetzen des Formulars)

this.myForm.patchValue({'userData': {'username': 'Manuel'}});

this.myForm.setValue({'userData': {

'username': 'Manuel',

'email': 'Manuel@test.com',

},

'gender': 'female',

'hobbies' : []});

## Umwandeln von Ausgaben mit Pipes

Oft werden Daten aus technischen Gründen nicht in einem Format gespeichert, welches vom Benutzer leicht zu interpretieren ist (z.B. Datum). Aus diesem Grund existieren in Angular sog. Pipes, mit welchem die Daten auf unkomplizierte Weise in ein verständliches Format überführt werden können. Diese können in Templates (nicht nur) mittels String-Interpolation verwendet werden indem der Name der Pipe – getrennt durch den Pipe-Operator | – an den Wert angehängt wird.

<strong>{{ server.name }}</strong> |

{{ server.instanceType | uppercase }} |

{{ server.started | date }}

app.component.html

Im Beispiel wird aufgrund der Verwendung der Pipe ‚uppercase‘ der Wert im Feld server.instanceType in Großbuchstaben ausgegeben.

### Build-In Pipes von Angular – Standardfälle

Für oft benötigte Fälle existieren in Angular vorgefertigte Pipes, welche direkt verwendet werden können. Eine Übersicht der in Angular verbauten Pipes ist in der offiziellen Dokumentation zu finden: <https://angular.io/api?query=pipe> . Nachfolgend finden Sie Beispiele von Pipes, die oft zum Einsatz kommen.

|  |  |  |
| --- | --- | --- |
| Pipe | Beschreibung | Parameter |
| uppercase | ALLES IN GROßBUCHSTABEN | - |
| lowercase | alles in kleinbuchstaben | - |
| titlecase | Anfangsbuchstaben Sind Großgeschrieben |  |
| date | Formatierung eines Datumsformats (konfigurierbar)  Mon Aug 09 1920 00:00:00 GMT+0100 (Mitteleuropäische Normalzeit) 🡺Aug 8, 1920 | format: z.B. fullDate oder eine frei definierte Folge erlaubter Zeichen  timezone: z.B. UTC+1  locale: z.B. en-US oder de\_DE  … |
| slice | Schneidet Teile aus einer Zeichenfolge:  {{'Hallo Welt' | slice:6:10}} 🡺 Welt | start : Beginn der Zeichenfolge  end: Ende der Zeichenfolge (optional) |

Pipes können auch konfiguriert werden, z.B. im Falle eines Datums. Die Syntax der Parameterübergabe besteht aus einem Doppelpunkt und dem Parameter in einfachen Anführungszeichen wenn es sich nicht um Zahlenwerte handelt. Weitere Parameter können mit zusätzlichen Doppelpunkten angehangen werden. Eine Weiterverarbeitung durch weitere Pipes ist durch das anhängen weiterer Pipes per Pipe-Operator (Reihenfolge beachten).

{{ variable | pipename:'param1':'param1' … | ... }}

z.B. {{'Hallo Welt' | slice:6:10 | uppercase}} 🡺 WELT

Vorsicht: {{ wert | date | uppercase}} statt {{wert | uppercase | date}}

### Internationalisierung

Insbesondere bei der Pipe für Datumswerte ist eine lokalisierte Darstellung von hoher Bedeutung für die User Experience. Standardmäßig ist nur die englische Locale in Angular verfügbar. Mittels der app.module.ts können jedoch auch andere Localen verfügbar gemacht werden, .z.B. deutsch.

Dies geschieht mit Hilfe der Methode registerLocaleData aus dem ‚@angular/common‘ Paket.

import { BrowserModule } from '@angular/platform-browser';

import { NgModule } from '@angular/core';

import { FormsModule } from '@angular/forms';

import { HttpModule } from '@angular/http';

import { AppComponent } from './app.component';

import { registerLocaleData } from '@angular/common';

import localeDe from '@angular/common/locales/de';

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

FormsModule,

HttpModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

registerLocaleData(localeDe, 'de');

app.module.ts

{{ server.started | date:'dd.MM.yyyy':'UTC+1':'de\_DE' }}

app.component.html

Weitere Informationen zum Einrichten von Localen sind in der Dokumentation zu finden: <https://angular.io/guide/i18n#i18n-pipes>

### Custom Pipes – Wenn’s komplizierter wird

Manchmal genügen die in Angular enthaltenen Pipes nicht aus. Mit Hilfe der Pipe Annotation können daher eigene Pipes erstellt werden. Im Beispiel entsteht eine Pipe, die einen übergebenen Text auf 10 Zeichen mit drei Punkten verkürzt.

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

@Pipe({

name: 'shorten'

})

export class ShortenPipe implements PipeTransform {

transform(value: any, ...args: any[]) {

if (value.length > 10) {

return value.substr(0, 10) + '...';

} return value;

}

}

shorten.pipe.ts

…

import { ShortenPipe } from './shorten.pipe';

@NgModule({

declarations: [

AppComponent,

ShortenPipe

],

…

})

export class AppModule { }

app.module.ts

<strong>{{ server.name | shorten }}</strong>

app.component.html

### Parametrisierung von Custom Pipes

Die Methode transform hat neben dem übergebenen Wert auch einen Parameter für einen oder mehrere Pipe-Parameter. Hier finden sich die Parameter wieder, die mit Hilfe des Doppelpunktes übergeben wurden.

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

@Pipe({

name: 'shorten'

})

export class ShortenPipe implements PipeTransform {

transform(value: any, limit: number) {

if (value.length > limit) {

return value.substr(0, limit) + '...';

} return value;

}

}

shorten.pipe.ts

<strong>{{ server.name | shorten:10 }}</strong>

app.component.html

### Im anderen Kontext - Pipes ohne String Interpolation

Pipes können auch in einem anderen Kontext als String Interpolationen verwendet werden, z.B. in Schleifen. Im Beispiel ist eine Pipe zum Filtern von Einträgen eines Arrays zu sehen. Hierbei kommen alle bereits gelernten Konzepte im Zusammenhang mit Pipes zum Einsatz. Der Parameter ‚value‘ enthält in diesem Fall das Array.

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

@Pipe({

name: 'filter'

})

export class FilterPipe implements PipeTransform {

transform(value: any, filterString: string, propertyName: string): any {

if (value.length === 0 || filterString === '' || filterString === undefined)

{

return value;

}

const resultArray = [];

for (const item of value) {

if (item[propertyName] === filterString) {

resultArray.push(item);

}

}

return resultArray;

}

}

filter.pipe.ts

export class AppComponent {

filteredStatus: string;

…

}

app.component.ts

<li class="list-group-item"

\*ngFor="let server of servers | filter:filteredStatus:'status'"

[ngClass]="getStatusClasses(server)">

app.component.html

Achtung: Der Filter wird nicht geändert, wenn sich das Array verändert, d.h. wenn neue Objekte hinzukommen oder wegfallen. Ein gegensätzliches Verhalten kann mit Hilfe des Parameters ‚pure‘ in der Pipe Definition erzwungen werden, was jedoch bei großen Datenmengen zu Einbußen in der Performance führen kann, da bei JEDER Datenänderung auf der Seite die Pipe neu ausgeführt wird.

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

@Pipe({

name: 'filter',

pure: false

})

export class FilterPipe implements PipeTransform {

…

}

filter.pipe.ts

### Asynchrone Pipes

Manchmal muss auf ein bestimmtes Ereignis gewartet werden, bis ein auszugebender Wert feststeht. In diesem Fall liegt i.d.R. ein Promise oder ein Observable vor, auf welches gewartet werden soll. Mittels der ‚async‘ Pipe kann im Template definiert werden, an welcher Stelle der Wert erscheinen soll. Die Aktualisierung erfolgt automatisch nach dem resolve/reject des Promise oder wenn ein Wert mittels eines Observables ankommt.

export class AppComponent {

appStatus = new Promise((resolve, reject) => {

setTimeout(() => {

resolve('stable');

},2000);

});

…

}

app.component.ts

<p>App Status: {{appStatus | async}}</p>

app.component.html

## Der Schritt nach draußen – HTTP Requests in Angular

Angular ist ein Framework, welches auf direktem Wege nur im Browser agieren kann, d.h. Zugriffe auf Datenbanken oder andere Prozesse sind nicht auf direktem Wege möglich. Dennoch stehen dem Entwickler mittels HTTP-Methoden Möglichkeiten zur Kommunikation mit externen Diensten zur Verfügung.

### Der klassische Ansatz (HttpModule)

Das HTTP-Modul wird als Service in die entsprechende Klasse injiziert. Hierbei kommt das Modul ‚HttpModule‘ aus dem Paket ‚@angular/http‘ zum Einsatz.

import { HttpModule } from '@angular/http';

…

imports: [

BrowserModule, AppRoutingModule, FormsModule, ReactiveFormsModule, HttpModule

],

app.module.ts

import {Headers, Http, Response} from '@angular/http';

import {Observable, throwError} from 'rxjs';

import {map, catchError} from 'rxjs/operators';

constructor(private http: Http) { // Injiziere HTTP

}

// Speichere Datensatz in Datenbank und übergebe Header-Optionen

storeServers(servers: any[]): Observable<any> {

const headers = new Headers({'Content-Type': 'application/json'});

return this.http.put('https://udemy-ng-http-a6889.firebaseio.com/data.json',

servers,

{headers: headers});

}

// Hole einfachen Datensatz ab. Wichtig ist, dass ein Observerable zurückgegeben wird, welches erst dann feuert, wenn jemand darauf subscribed

getServers(): Observable<any> {

return this.http.get('https://udemy-ng-http-a6889.firebaseio.com/data.json');

}

// Hole Datensatz ab und wandele ihn mittels des rxjs-Operators map in seine json-Form. Error-Handling erfolgt mittels des Operators catchError

getServersTransformed(): Observable<any> {

return this.http.get('https://udemy-ng-http-a6889.firebaseio.com/data.json')

.pipe(map(

(response: Response) => {

// Umwandlung nach JSON nötig

return response.json();

}

))

.pipe(catchError(

(error: Response) => {

console.log(error);

return throwError('Something went wrong: ' + error);

}

))

;

}

servers.service.ts

Wichtig ist hierbei, dass jede HTTP-Methode ein Observable zurückliefert, welches erst dann Daten liefert, wenn darauf gelauscht wird. Das bedeutet auch, dass die Anfrage erst dann ausgeführt wird, wenn mittels subscribe() aktiv auf ein Ergebnis gelauscht wird. Mittels Parameter können Optionen wie z.B. Header Informationen mitgegeben werden.

Mittels der RXJS Operatoren ‚map‘ und ‚catchError‘ wird das Ergebnis bzw. einer mögliche Fehlermeldung weiter verarbeitet. Im Beispiel wird das Ergebnis der Abfrage mittels der ‚json()‘ Methode aus dem Response Objekt extrahiert und die Fehlermeldung mittels eines erklärenden Texts erweitert.

export class AppComponent {

constructor(private service: ServerService) {

}

// Speichere Observable zur Nutzung mit einer async Pipe zwischen

serversAsync = this.service.getServersTransformed();

servers = [];

onSave() {

this.service.storeServers(this.servers).subscribe((res) => {

console.log(res);

}, (error) => { console.log(error);

});

}

// Subscribe das Observable und führe Error-Handling durch

onGetServers() {

this.service.getServersTransformed().subscribe((data: any[]) => {

this.servers = data;

}, (error) => {

console.log(error);

});

}

}

app.component.ts

<div class="container">

<div class="row">

<div class="col-xs-12 col-sm-10 col-md-8 col-sm-offset-1 col-md-offset-2">

<input type="text" #serverName>

<div class="form-group">

<button class="btn btn-primary" (click)="onSave()">Save Servers</button>

<button class="btn btn-primary" (click)="onGetServers()">Get Servers</button>

</div>

<hr>

<ul class="list-group" \*ngFor="let server of serversAsync | async">

<li class="list-group-item">{{ server.name }} (ID: {{ server.id }})</li>

</ul>

</div>

</div>

</div>

app.component.html

Auch hier kann problemlos die ‚async‘ Pipe zum Einsatz kommen, um direkt beim Betreten der Site die Daten asynchron nachzuladen.

### Die modernere Alternative (HttpClientModule)

Das HTTPClient-Modul wird ebenfalls als Service in die entsprechende Klasse injiziert und funktioniert grundsätzlich sehr ähnlich wie der klassische Ansatz. Hierbei kommt das Modul ‚HttpClientModule‘ aus dem Paket ‚@angular/common/http‘ zum Einsatz. Das HttpClient-Modul bietet feingranulare Einstellungsmöglichkeiten z.B. in Bezug auf das erwartete Ergebnis einer Abfrage.

#### Vewendung des HttpClient

import { HttpClientModule} from '@angular/common/http';

…

imports: [

BrowserModule, AppRoutingModule, FormsModule, ReactiveFormsModule, HttpClientModule

]

app.module.ts

constructor(private httpClient: HttpClient, private recipeService: RecipeService) {

}

fetchRecipesWithHttpClient() {

// Direkter Zugriff auf die Body-Eigenschaft der Response wegen HttpClient

this.httpClient.get<Recipe[]>('https://ng-recipe-book-8f3cd.firebaseio.com/recipes.json'

/\* Man kann in den Optionen auch angeben, was extrahiert werden soll. Möchte man die komplette

Response, so muss man 'response' observen

,{

observe: 'body',

responseType: 'json'

}\*/

)

.pipe(map(

(recipes) => {

for (const recipe of recipes) {

if (!recipe['ingredients']) {

recipe['ingredients'] = [];

}

}

return recipes;

}

))

.pipe(

catchError((error: Response) => {

return throwError(error);

}))

.subscribe((recipes: Recipe[]) => {

this.recipeService.setRecipes(recipes);

});

}

app.component.ts

Ein wichtiger Unterschied besteht darin, dass beim Einsatz von HttpClient die Daten direkt zur Verfügung stehen, d.h. man erhält nicht die Response, sondern die Payload (body) des Pakets und es entfällt der Aufruf der Methode ‚json()‘. Die HTTP-Methoden selbst sind generisch, d.h. man kann bereits festlegen, welches Format erwartet wird.

Wird die Response benötigt, so muss dies im Header mittels der Eigenschaft ‚observe‘ und dem Wert ‚response‘ spezifiziert werden. Standardmäßig ist diese auf den Wert ‚body‘ voreingstellt. Mit Hilfe der ‚responseType‘ Eigenschaft kann auch der Typ der erwarteten Response angegeben werden. Standardmäßig ist dies ‚json‘.

#### Statusupdates mit HttpEvents und benutzerdefinierte Requests mit HttpRequest

Werden Statusnachrichten z.B. über den erfolgreichen Versand einer Nachricht oder des Upload / Download Status von Dateien benötigt, kann das Eventsystem des HttpClient verwendet werden. Hierbei wird die ‚observe‘ Eigenschaft der Options auf den Wert ‚events‘ gesetzt. Das Ergebnis einer Anfrage ist dann vom generischen Typ HttpEvent.

saveRecipesWithEvents() {

return this.httpClient.put('https://ng-recipe-book-8f3cd.firebaseio.com/recipes.json',

this.recipeService.getRecipes(),

{observe: 'events'});

}

servers.service.ts

saveDataWithEvents() {

this.dataStorageService.saveRecipesWithEvents().subscribe((res: HttpEvent<Object>) => {

if (res.type === HttpEventType.Sent) {

console.log('Message was sent');

}

}, (error) => {

console.log(error);

});

}

app.component.ts

Ein ähnliches Verhalten kann mit der Eigenschaft ‚reportProgress‘ erreicht werden. Hierbei muss jedoch ein benutzerdefinerter Request erstellt werden.

saveRecipesWithCustomRequest() {

const req = new HttpRequest('PUT', 'https://ng-recipe-book-8f3cd.firebaseio.com/recipes.json',

this.recipeService.getRecipes(),

{reportProgress: true});

return this.httpClient.request(req);

}

servers.service.ts

Auch dieses Observable sendet alle Events aus, auf welche reagiert werden kann.

#### Header-Eigenschaften

Manche APIs benötigen spezielle Header Informationen, wie z.B. Authentifizierungsdaten oder einen API Key. Diese Daten können am HTTP Header mitgesendet werden. Dies geschieht – wie schon von der Eigenschaft ‚observe‘ bekannt – über die Optionen eines HTTP Requests.

return this.httpClient.put('https://ng-recipe-book-8f3cd.firebaseio.com/recipes.json',

this.recipeService.getRecipes(),

{observe: 'events',

/\* set() setzt eine Headereigenschaft, append() fügt eine

neue hinzu \*/

headers: new HttpHeaders().set('Authorization', 'MYAPIKEY')

});

servers.service.ts

#### Query Parameter

In manchen Fällen ist es sinnvoll, Parameter direkt in der Query - also als Teil der URL – mitzusenden. Dies führt jedoch schnell zu sehr langen und unübersichtlichen URLS. Aus diesem Grund können Parameter auch im Header übergeben werden. Das Handling verhält sich ähnlich wie die bereits erwähnten Header Eigenschaften.

return this.httpClient.put('https://ng-recipe-book-8f3cd.firebaseio.com/recipes.json',

this.recipeService.getRecipes(),

{observe: 'events',

/\* set() setzt eine Headereigenschaft, append() fügt eine

neue hinzu \*/

params: new HttpParams().set('auth', 'MYTOKEN')

});

servers.service.ts

#### Interceptors - Ein und ausgehende Datenpakete bearbeiten

Es ist mühsam und redundant, bei jedem einzelnen Request wiederkehrende Parameter neu setzen zu müssen. Aus diesem Grund können sog. Interceptors zum Einsatz kommen, welche auf ein und ausgehende Datenpakete lauschen und diese entsprechend bearbeiten können. Wichtig ist hierbei, dass die Requests vom Typ HttpRequest immutable sind, d.h. sie können nicht bearbeitet werden. Aus diesem Grund muss jeder Request zuvor geclont werden. Der hierfür verwendeten Methode clone() können – wie schon von den Methoden bekannt – Optionen mitgegeben werden. Hierbei ist jedoch zu beachten, dass Parameter oder Header vom alten Request verwendet werden müssen, anstelle neue zu erzeugen.

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

import { tap } from 'rxjs/operators';

import {

HttpEvent, HttpInterceptor, HttpHandler, HttpRequest

} from '@angular/common/http';

@Injectable()

export class LoggingInterceptor implements HttpInterceptor {

intercept(req: HttpRequest<any>, next: HttpHandler) {

// Request ist immutable => Kopie erstellen und Parameter übergeben. WICHTIG: alte Requestparameter verwenden !!

const copiedRequest = req.clone({params: req.params.set('auth', 'MYTOKEN')});

return next.handle(copiedRequest).pipe(tap((res) => console.log('Logging interceptor: ', res));

// oder null, wenn Request nicht weitergesendet werden soll

}

}

servers.service.ts

Soll z.B. das Ergebnis geloggt werden so kann dies mit dem RXJS Operator tap() geschehen, da dieser – anders als subscribe – den Request im Observable nicht auslöst.

providers: [ShoppingListService,

RecipeService,

DataStorageService,

{provide: HTTP\_INTERCEPTORS, useClass: LoggingInterceptor, multi: true}], // multi, damit mehrere Interceptors verwendet werden können

app.module.ts

## Animationen

CSS3 unterstützt nativ die Verwendung von Animationen. Zusätzlich existiert auch eine Möglichkeit zur Definition von Animationen mit Angular Syntax. Sie werden innerhalb von Komponenten definiert und als Eigenschaft an die jeweiligen HTML Elemente gehangen.

### Trigger, States und Styles

Animationen werden in Angular als Übergang zwischen einem Start und Endzustand über einen festzulegenden Zeitraum definiert, wobei jeder Zustand einen eigenen Style besitzt.

Animationen werden innerhalb der Komponentenbeschreibung mit Hilfe der Eigenschaft „animation“ definiert. Diese akzeptiert ein Array von „trigger“ Elementen, deren selbst definierbarer Bezeichner als Schlüsselwort zur Verwendung an HTML dient. Zur Festlegung des Zustands (state) existier die Möglichkeit, eine Variable mittels Property-Binding an den Trigger zu binden. Sobald sich der Wert der Zustandsvariable ändert, wird der Übergang ausgelöst (getriggert) und die Animation ausgeführt.

<div style="width:100px; height: 100px;" [@divState]="state"></div>

app.component.html

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

animations: [

// Animation eines Div-Containers mit zwei States

trigger('divState', [

state('normal',

style({'background-color': 'red',

transform: 'translateX(0)'})

),

state('highlighted',

style({'background-color': 'blue',

transform: 'translateX(100px)'})

),

transition('normal <=> highlighted', animate(300))

]),

…]

export class AppComponent {

state = 'normal';

…

app.component.ts

Im Beispiel verändert ein div-Container seine Position und Farbe innerhalb von 300ms, wenn eine Statusänderung eintritt. Aufgrund des beidseitigen Pfeil-Operators (<=>) ist es unerheblich ob der Start- oder Endzustand zuvor „normal“ oder „highlighted“ war. In beiden Fällen wir die Animation ausgeführt.

### Wildcard und komplexere Animationen

Neben den einfachen Zustandsübergängen mit definierten Bezeichnern existieren zwei weitere Möglichkeiten. Die erste ist der Wildcard-Operator (\*), mit welchem ein beliebiger Status angegeben werden kann. Dies funktioniert in beide Richtungen.

<div style="width:100px; height: 100px;" [@wildState]="wildState"></div>

app.component.html

// Animation eines Div-Containers mit drei States und Wildcard-Operator mit Zwischenschritten

trigger('wildState', [

state('normal',

style({'background-color': 'red',

transform: 'translateX(0px) scale(1)'})

),

state('highlighted',

style({'background-color': 'blue',

transform: 'translateX(100px) scale(1)'})

),

state('shrunken',

style({'background-color': 'blue',

transform: 'translateX(0px) scale(0.5)'})

),

transition('highlighted => normal', animate(700)),

transition('normal => highlighted', animate(400)),

transition('shrunken <=> \*', [

// Setze Farbe auf Orange, ...

style({

'background-color': 'orange'

}),

// Verändere den Rahmenradius

animate(1000, style({

'border-radius': '50px'

})),

// ... und animiere über eine halbe Sekunde bis zum Endstatus

animate(500)

])

])

…

export class AppComponent {

wildState = 'normal';

…

app.component.ts

Es können innerhalb eines Überganges mehrere, aufeinanderfolgende Schritte definiert werden. Diese werden innerhalb eines Arrays im „transition“ Aufruf festgelegt. Hierbei können sowohl direkte Styles als auch Style-Übergänge über einen Zeitraum definiert werden.

### Animationen beim Hinzufügen und Entfernen von Elementen und Aufteilen von Animationen mittels Keyframes

// Animation von Listenelementen beim Hinzufügen und Entfernen

trigger('list1', [

// End-Stil nach dem Einblenden

state('\*', style({opacity: 1, transform: 'translateX(0px)'})),

// Beim Einblenden wird ein Start-Stil definiert, um ein Einfliegen von links zu simulieren.

transition('void => \*', [style({opacity: 0, transform: 'translateX(-100px)'}), animate(400)]),

// Beim Ausblenden wird ein Endstil definiert, welcher vor dem Entfernen erreicht werden soll

transition('\* => void', animate(300, style({opacity: 0, transform: 'translateX(-100px)'})))

]),

// Animation von Listenelementen beim Hinzufügen und Entfernen (mit Keyframes)

trigger('list2', [

state('\*', style({opacity: 1, transform: 'translateX(0)'})),

transition('void => \*', [

// Styleveränderungen während der Transition, also in diesem Fall innerhalb einer Sekunde

animate(1000, keyframes([

style({transform: 'translateX(-100px)', opacity: 0, offset: 0}),

style({transform: 'translateX(-50px)', opacity:0.5, offset: 0.3}),

style({transform: 'translateX(-20px)', opacity:1, offset: 0.8}),

style({opacity: 1, offset: 1, transform: 'translateX(0px)'})

]))

]

),

transition('\* => void', animate(300, style({opacity: 0, transform: 'translateX(-100px)'})))

]),

app.component.ts

## Gruppierung von Animationen

// Animation von Listenelementen beim Hinzufügen und Entfernen mit Grouping

trigger('list3', [

state('\*', style({opacity: 1, transform: 'translateX(0px)'})),

transition( 'void => \*',

[

style({

opacity: 0,

transform: 'translateX(-100px)'}),

animate(400)

]),

transition( '\* => void',

[

// Gruppierung der beiden Animationen, damit diese gleichzeitig ausgeführt werden

group([

animate( 300,

style({

'color': 'red'}

)

),

animate( 300,

style({

opacity: 0,

transform: 'translateX(-100px)'}

)

)

])

])

]),

app.component.ts

Beispiel: Gleichzeitiges Einfärben und Ausfaden des Listenelements

## Eventhandler in Animationen

<div style="width:100px; height: 100px;"

[@divState]="state"

(@divState.start)="onAnimationStarted($event)"

(@divState.done)="onAnimationEnded($event)">

</div>

app.component.html

// Animation eines Div-Containers mit zwei States

trigger('divState', [

state('normal',

style({'background-color': 'red',

transform: 'translateX(0)'})

),

state('highlighted',

style({'background-color': 'blue',

transform: 'translateX(100px)'})

),

transition('normal <=> highlighted', animate(300))

]),

…

export class AppComponent {

state = 'normal';

onAnimationStarted(event) {

console.log(event);

}

onAnimationEnded(event) {

console.log(event);

}

app.component.ts

## Debugging von TypeScript Code

TypeScript Code kann mit Hilfe der Chrome Developer Tools (F12 in Google Chrome) zur Laufzeit debugged werden. Hierzu müssen die Tools geöffnet werden. Dort wird anstelle von „Elements“ der Reiter „Sources“ ausgewählt und dort in der Datei „main.bundle.js“ nach der Stelle gesucht, die am Besten zum Problem passt. Der dort dargestellte Code ist NICHT TypeScript, sondern der generierte JavaScript Code. Setzt man dort jedoch einen Breakpoint, so ist der TypeScript Code einsehbar. Dies liegt an den erstellten Source Maps, in welchen der korrespondierende TypeScript Code hinterlegt wird.

* Eine bessere Lösung findet sich im „webpack://“ Verzeichnis. Klickt man dieses auf, so kann man in den „./src/app“ Pfad direkt auf den TypeScript Code zugreifen.
* Ein gutes Tool zur Analyse von Angular Apps stellt die Chrome Erweiterung „Augury“ dar. Diese kann z.B. alle Components und deren Eigenschaften auflisten. Auch Abhängigkeiten wie z.B. Injections können damit dargestellt werden.

## ES6 (ECMAScript) Features

|  |  |  |
| --- | --- | --- |
| Name | Beschreibung | Syntax |
| Spread-Operator | Wandelt ein Array in eine Liste um. Das ist z.B. bei push() auf dem Array Typ interessant, um diesem mehrere Items gleichzeitig hinzuzufügen. | **this**.**ingredients**.push(...ingredients);  anstelle von  **for** (**const** ingredient **of** ingredients) {  **this**.add(ingredient); } |