1

Komponenten und Datenbindung

[Datenbindung in Angular 1](#_Toc480138500)

[Eigene Komponenten mit Datenbindung 8](#_Toc480138501)

[Life Cycle Hooks 16](#_Toc480138502)

Eine Angular-Anwendung besteht aus Komponenten, die wiederum aus Komponenten bestehen. Es ergibt sich somit ein Komponentenbaum. Somit kann eine komplexe Anwendung auf mehrere einfache, wiederverwendbare und testbare Teile heruntergebrochen werden.

Während die Einführung zu Angular bereits eine erste Komponente beschrieben hat, zeigt dieses Kapitel, wie man eine solche Komponente in mehrere zerlegen kann, die über Datenbindung miteinander kommunizieren. Dabei handelt es sich auch um das Grundprinzip, über das sich der Komponentenbaum einer Anwendung ergibt.

# Datenbindung in Angular

Bevor das vorliegende Kapitel zeigen kann, wie Komponenten über Datenbindung kommunizieren, geht dieser Abschnitt darauf ein, wie Datenbindung in Angular überhaupt funktioniert.

## Rückblick auf AngularJS 1.x

Um die Architekturentscheidungen hinter der Datenbindung in Angular zu verstehen, lohnt sich ein Blick auf den Vorgänger, AngularJS 1.x. Hier konnte alles an alles gebunden werden. Um dies zu veranschaulichen, zeigt

zwei Datenmodelle sowie eine Direktive, die aneinander gebunden wurden. Die Direktiven entsprechen in diesem Beispiel den heutigen Komponenten.

Abbildung 1: Zyklen in AngularJS 1.x

Durch die vielen wechselseitigen Abhängigkeiten konnte eine Änderung zu weiteren Änderungen führen und diese konnten wiederum weitere Änderungen nach sich ziehen. Deswegen musste AngulaJS 1.x auch im Kreis laufen. Dieses im Kreis laufen war auch als Digest-Cycle bekannt und war natürlich der Performance alles andere als zuträglich. Obwohl AngularJS 1.x in vielen Fällen schnell genug war, ist man durch dieses Verhalten das eine oder andere Mal – abhängig von der Anwendungsarchitektur – in Performancefallen getappt und da ist man dann nur wieder schwer herausgekommen.

Bei Angular (ab Version 2) ist die Situation anders: Hier ist die Anwendung ein Komponentenbaum. Somit ergibt sich eine hierarchische Struktur, die das Einführen einiger einfacher Regeln ermöglicht (Abbildung 2). Diese Regeln sind unter anderem der Schlüssel für die extrem gute Performance der Neuauflage. Die nächsten Abschnitte gehen darauf ein.

Abbildung 2: Hierarchische Struktur einer Angular-Anwendung

## Property Bindings

Für Property Bindings gilt in Angular, dass sie immer nur Daten von einer Parent- zu einer Child-Komponente weitergeben. Anders ausgedrückt bedeutet das, dass Child-Komponenten im Rahmen der Datenbindung nicht ihren Parent verändern dürfen (Abbildung 3). Die Daten fließen hier also von oben nach unten.

Abbildung 3: Die Daten fließen bei Property-Bindings von oben nach unten

Der Abhängigkeitesgraph, der bei AngularJS 1.x noch zyklen enthalten konnte, ist nun ein Baum – also frei von Zyklen. Somit muss Angular auch nicht mehr im Kreis laufen. Vielmehr reicht es, den Baum ein einziges Mal zu traversieren, um ihn mit der UI abzugleichen – Oder anders ausgedrückt: Man benötigt lediglich einen einzigen Digest im Sinne von AngularJS 1.x. Da Angular für das Traversieren des Baums Code generiert, der sich gut von JavaScript-Engines optimieren lässt, ist dieser Vorgang äußerst schnell. Standardmäßig findet diese Codegenerierung beim Start der Anwendung statt. Dank der in Kapitel 18 beschriebenen Ahead-of-Time-Kompilierung (AOT-Kompilierung) kann sich bereits der Build-Prozess darum kümmern.

Hinweis Beginn

Um das Property-Binding zu optimieren, kann man das Framework wissen lassen, welcher Teil des Baums sich geändert hat. In diesem Fall beschränkt sich Angular auch nur auf den betroffenen Teilbaum. Da es sich hierbei um eine Optimierungstechnik handelt, die über die Grundlagen hinaus geht, finden sich Informationen dazu erst im weiter hinten platzierten Kapitel 12.

Hinweis Ende

## Event Bindings

Event-Bindings definieren Handler für Ereignisse, die in Kind-Komponenten auftreten. Beispiele dafür sind das Click-Event aus der Einführung zu Angular. Die Schaltfläche löst es aus, behandelt wurde es jedoch im Handler der übergeordneten *FlightSearchComponent*.

Beim Auslösen eines Events kann die Kind-Komponente einen Wert angeben, den der Handler im Parent erhält. Dies dient der genaueren Beschreibung des aktuellen Ereignisses. Somit fließen bei Event-Bindings Informationen von unten nach oben (Abbildung 4).

Abbildung 4: Die Daten fließen bei Event-Bindings von unten nach oben

Das Schöne an Event-Bindings ist, dass sie sehr billig sind: Das Auslösen eines Event-Handlers besteht mehr oder weniger nur im Aufruf einer festgelegten Methode. Aus diesem Grund benötigt Angular dafür, um abermals die Nomenklatur aus AngularJS 1.x zu bemühen, keinen einzigen Digest.

Allerdings kann ein Event den Zustand der Anwendung verändern. Jemand klickt auf Auswählen und plötzlich befindet sich ein neuer Flug im Warenkorb oder jemand klickt auf Suchen und ein paar Augenblicke später liegen neue Flüge vor, die es anzuzeigen gilt. Wie Angular damit umgeht, zeigt der nächste Abschnitt.

## Zusammenspiel von Property- und Event-Bindings

Damit die von Event-Handler an den Komponenten durchgeführten Änderungen in der UI präsentiert werden, greifen bei Angular die beiden betrachteten Arten der Datenbindung ineinander. Das lässt sich durch einen Zustandsautomaten sehr gut veranschaulichen (Abbildung 5).

Abbildung 5: Datenbindung als Zustandsautomat

Nach dem Initialisieren der Anwendung ist sie bereit und wartet auf Ereignisse. Hierbei kann es sich um Benutzer-Ereignisse (z. B. *click* oder *keypress*), Zeitereignisse (z. B. *timeout*) oder Datenereignisse (z. B. das empfangen serverseitiger Daten) handeln. Um diese Ereignisse zu erkennen, klinkt sich Angular mit einem Mechanismus, der sich *zone.js* nennt, beim Start in sämtliche Event-Handler ein.

Tritt ein Ereignis auf, führt Angular die dafür registrierten Event-Handler aus. Ein Event-Handler kann weitere Events auslösen. Auf diese weise kann eine Anwendung Informationen den Komponentenbaum nach oben transportieren.

Sind alle Event-Handler ausgeführt worden, führt Angular ein Property-Binding durch. Das ist notwendig, weil jedes Event gebundene Daten tendenziell verändern kann. Dazu traversiert es den gesamten Komponentenbaum ein einziges Mal. Wie schon erwähnt, ist dieser Vorgang bei Angular gut optimiert und somit in der Regel auch sehr schnell. Darüber hinaus existieren Optimierungsmöglichkeiten hierfür (siehe Kapitel XYZ).

Danach ist die Anwendung abermals bereit und das Spielchen beginnt von vorne. So folgt auf jede Event-Phase ein Property-Binding. Somit findet genau ein Digest anstatt einer Vielzahl an Digests bei AngularJS 1.x statt. Wichtig ist hier auch, dass ein Property-Binding keine Events auslösen darf. Das würde ja einer Änderung des Parent gleichkommen und sowas ist – wie diskutiert – per Definition verboten. Somit verhindert Angular aufgrund seiner Architektur, dass mehr als ein Digest zu einem Zeitpunkt stattfindet.

## Bindings im Template

Wie schon in der Einführung zu Angular erwähnt, werden die beiden Arten von Bindings durch eine jeweils eigene Schreibweise im Template ausgedrückt. Während für Property-Bindings eckige Klammern zum Einsatz kommen, greift man bei Event-Bindings zu runden (Listing 1). Somit ist auf dem ersten Blick ersichtlich, welche Binding vorliegt.

<button [disabled]="!from || !to" (click)="search()">  
 Search  
</button>

<table>

<tr \*ngFor="let flight of flights">

<td>{{flight.id}}</td>

<td>{{flight.date}}</td>

<td>{{flight.from}}</td>

<td>{{flight.to}}</td>

<td><a href="#" (click)="selectFlight(flight)">Select</a></td>

</tr>

</table>

Listing 1: Template mit Bindings

Sogar die Syntax mit den doppelt geschweiften Klammern ist genau genommen nichts anderes als eine Kurzschreibweise für ein Property-Binding. Im betrachteten Fall ein Property-Binding an die DOM-Property text-content, welche den textuellen Inhalt eines Knoten wiederspiegelt. Die Schreibweise

<td>{{ flight.id }}</td>

ist somit gleichbedeutend mit

<td [text-content]="flight.id"></td>

Two-Way-Bindings wurden in diesem Beispiel ausgespart. Warum, erklärt der nächste Abschnitt.

## Two-Way-Bindings

Wie schon öfters erwähnt, kennt Angular Event-Bindings und Property-Bindings. Und beide Bindingarten spielen zusammen. Da stellt sich nun die Frage, wo die Two-Way-Bindings geblieben sind. Gerade in formular-basierten Anwendungen muss man ja häufig den Formularzustand mit einem Objektmodell abgleichen.

Die Antwort hierauf lautet, dass ein Two-Way-Bindings in Angular nichts anders, als eine Kombination aus einem Property-Binding und einem Event-Binding ist: Das Property-Binding transportiert den jeweiligen Wert von der Komponente ins Eingabefeld und bei einer Änderung durch den Benutzer transportiert das Event-Binding die Daten wieder retour in die Komponente. In erster Näherung könnte als ein Two-Way-Binding wie folgt formuliert werden:

<input [ngModel]="from" (ngModelChange)="update($event)">

Dieses Beispiel nutzt *ngModel* als Property-Binding und das dazugehörige *ngModelChange* als Event-Binding. Letzteres wird bei jeder Änderung am Datenfeld aktiv und erhält über *$event* den geänderten Wert. Ändert der Benutzer beispielsweise Hamburg auf Frankfurt um, befindet sich in *$event* der neue Wert Frankfurt. Diesen Wert übergibt es an die Methode update, welche nun die Ausgangsvariable *from* aktualisiert:

update(f: Flight) {   
 this.selectedFlight = f;   
}

Das ganze lässt sich ein wenig abkürzen, indem der Code des Event-Handlers direkt im Event-Binding hinterlegt wird:

<input [ngModel]="from" (ngModelChange)="from = $event">

Noch kürzer wird es mit der bereits vorgestellten Banana-in-a-Box-Syntax:

<input [(ngModel)]="from">

Hierbei gilt jedoch zu beachten, dass diese Grammatik nichts anders als die Kurzschreibweise für die zuvor betrachtete explizite Form ist. Um daraus ein Property-Binding genieren, streift Angular einfach die runden Klammern ab. Das Event-Binding wird durch Anhängen der Endung Change hergestellt. Aus *[(ngModel)]* wird somit *(ngModelChange)*. Daneben geht Angular davon aus, dass solche Events den geänderten Wert als *$event* erhalten. Diesen schreibt es in die Ausgangsvariable zurück.

Durch diese Architektur benötigt Angular selbst für das aktualisieren von Two-Way-Bindings pro Änderung maximal einen Digest. Außerdem sollte man diese Konvention im Hinterkopf haben, wenn man eigene Komponenten schreibt. Sollten die ein Two-Way-Binding anbieten, muss man sich genau an diese Konventionen halten. Der nächste Abschnitt geht darauf anhand eines Beispiels ein.

# Eigene Komponenten mit Datenbindung

Nachdem nun besprochen wurde, wie Datenbindung unter Angular funktioniert, geht dieser Abschnitt auf die Schaffung eigener Komponenten, die über Bindings mit der Außenwelt kommunizieren, ein. Dazu wird eine Komponente geschaffen, die so einfach wie möglich aber so komplex wie nötig ist, um die vorhin diskutierten Konzepte zu zeigen. Es handelt sich dabei um eine Komponente, welche Flüge in Form von Karten präsentiert.

Abbildung 6: FlightCardComponent

Solche Karten sind derzeit sehr üblich, zumal sie ein flexibles („responsive“) Design erlauben: Steht am Endgerät viel Platz zur Verfügung, kann eine Anwendung mehrere Karten nebeneinander anzeigen. Steht wenig Platz zur Verfügung, zeigt die Anwendung die Karten untereinander an.

Jede Karte kann ausgewählt werden. Wurde sie ausgewählt, erhält sie einen orangen Hintergrund; ansonsten einen blauen. Außerdem sollen alle ausgewählten Flüge im Warenkorb präsentiert werden. Dazu wird der Warenkorb auf eine *Map*, welche die Ids der Flüge auf einen *boolean* abbildet, abgeändert:

public basket = new *Map*<number, boolean>();

Um zu prüfen, ob sich ein Flug im Warenkorb befindet, muss die Anwendung also nur prüfen, ob der Basket an der Stelle der FlugId truthy ist:

let inBasket = this.basket[7]; // 7 ist eine FlugId

Zur Visualisierung des Warenkkorbs kommt aus Gründen der Vereinfachung abermals die JSON-Pipe zum Einsatz:

{{ basket | json }}

Das Ganze gestaltete sich dann wie in Abbildung 7 gezeigt.

Abbildung 7: Ausgabe des Warenkorbes

## Komponente mit Property Binding

Die hier besprochene Karte soll über Property-Bindings zwei Informationen vom Parent übergeben bekommen: Den anzuzeigenden Flug und die Information, ob sie ausgewählt wurde. Für erstere weist die Komponente eine Eigenschaft item und für zweitere eine Eigenschaft *selected* auf:

<div \*ngFor="let f of flights">

<flug-card [item]="f" [selected]="basket[f.id]">

</flug-card>

</div>

Um alle gefundenen Flüge auszugeben, iteriert das betrachtete Beispiel über die Auflistung flights und gibt pro Eintrag eine Karte aus.

Somit kann man sich das einbinden einer Komponente, wie den Aufruf einer Funktion, welche Parameter übergeben bekommt und ein Stück UI rendert, vorstellen. Eine andere Metapher für eine Komponente ist ein elektronisches Bauteil, wie ein Chip: Er ist über Eingänge mit der Außenwelt verdrahtet und bekommt auf diese Weise die nötigen Informationen (Abbildung 8).

Abbildung 8: Die Komponente flight-card

Im hier betrachteten Fall nimmt der Eingang *item* den jeweiligen Flug sowie der Eingang *selected* den entsprechenden *boolean* aus dem Warenkorb entgegen.

Jetzt stellt sich natürlich die Frage, wie man mit Angular solche Eingänge darstellt. Der nächste Abschnitt geht darauf ein.

### Implementierung der Komponente mit Property Bindings

Die Implementierung der in den letzten Abschnitten besprochenen Komponente besteht zunächst mal aus einer Klasse mit einem *Component*-Dekorator. Dieser erhält einen Selektor sowie einen Verweis auf ein Template. Das ist soweit nichts Neues. Neu ist allerdings der *Input*-Dekorator. Er dekoriert sämtliche Eigenschaften, die die Komponente von ihrem Parent entgegennimmt (Listing 2)

import { Component, EventEmitter, Input, Output } from '@angular/core';

import { Flight } from '../../entities/flight';

@Component({

selector: 'flight-card',

templateUrl: './flight-card.component.html'

})

export class FlightCardComponent {

@Input() item: Flight;

@Input() selected: boolean;

select() {

this.selected = true;

}

unselect() {

this.selected = false;

}

}

Listing 2: Implementierung einer Komponente mit Property-Bindings

Außerdem weist sie zwei Methoden auf, welche ihr Template aufruft: *select* wählt die Karte aus und *deselect* hebt diese Auswahl wieder auf. Das Template dieser Komponente prüft zunächst, ob die Karte selektiert wurde. Ist dem so, erhält sie per *ngStyle* die Hintergrundfarbe *orange*; ansonsten *lightsteelblue* (Listing 3).

<div style="padding:20px;"

[ngStyle]="{'background-color': (selected) ? 'orange' : 'lightsteelblue' }">

<h2>{{item.from}} - {{item.to}}</h2>

<p>Flugnr. #{{item.id}}</p>

<p>Datum: {{item.date | date:'dd.MM.yyyy HH:mm'}}</p>

<p>

<button \*ngIf="!selected" class="btn btn-default" (click)="select()">Select</button>

<button \*ngIf="selected" class="btn btn-default" (click)="deselect()">Unselect</button>

</p>

</div>

Listing 3: Template der flight-card

Das Template gibt ein paar Daten des aktuellen Fluges aus und hat am unteren Ende zwei Schaltflächen, die *select* bzw. *deselect* anstoßen. Zu einem Zeitpunkt zeigt die Karte jedoch nur eine der beiden Schaltflächen an. Das hängt davon ab, ob die Karte gerade ausgewählt ist.

### Komponente registrieren und aufrufen

Um die Komponente verwenden zu können, ist sie bei einem Angular-Modul zu registrieren. Zur Vereinfachung kommt hier das *AppModule* zum Einsatz (Listing 4) Das Zusammenspiel mehrerer Module wird später Kapitel 7 betrachten.

@NgModule({

[...],

declarations: [

FlightSearchComponent,

FlightCardComponent,

[...]

]

})

export class AppModule {

}

Listing 4: Komponente registrieren

Danach erhält das gesamte Modul Zugriff auf die Komponente und lässt sich z. B. zur Präsentation gefundener Flüge verwenden. Hierzu greift dieses Beispiel auf die *FlightSearchComponent* aus Kapitel 3 zurück. Zunächst erhält sie den beschrieben Warenkorb. Dann ist noch die Schleife, welche die Suchergebnisse listet, abzuändern, sodass sie die Karten rendert (Listing 5).

<div \*ngFor="let f of flights">

<flight-card [item]="f" [selected]="basket[f.id]">

</flight-card>

</div>

Listing 5: Einbinden der FlightCardComponent

Wie besprochen, erhält diese Komponente den aktuellen Flug und den Boolean aus dem Warenkorb. Die Anwendung sollte nun wie eingangs gezeigt die gefundenen Flüge als Karten präsentieren.

Die Karten lassen sich auch über die präsentierten Schaltflächen aus- und abwählen. Ein kleines Problem fällt dabei allerdings auf: Angular aktualisiert den Warenkorb nicht. Hierzu müsste die *FlightCardComponent* ihren Parent, der den Warenkorb verwaltet, mit einem Ereignis benachrichtigen. Wie das geht, erläutert der nächste Abschnitt.

Hinweis Beginn

Um mehrere Karten nebeneinander zu präsentieren, kann man zum Spaltenlayout von Bootstrap greifen. Es ist für responsive Designs – also für Designs, die sich an unterschiedliche Auflösungen anpassen – gedacht. Dazu unterteilt es eine Seite in zwölf gedachte Spalten und die Anwendung weist jedem Element eine bestimmte Anzahl an Spalten zu. Dabei kann sie zwischen sehr kleinen (extra small, xs), kleinen (small, sm), mittleren (medium, md) und großen (large, lg) Bildschirmen unterscheiden. Beispiele für diese vier Größeneinheiten sind Handys, Tabletts, kleine Laptops und Desktop-Geräte. Hierbei handelt es sich jedoch nur um Näherungen, denn schlussendlich kommt es auf die zur Verfügung stehende Auflösung an.

Beispielsweise könnte man nun angeben, dass eine Karte bei sehr kleinen Geräten (xs) alle zwölf Spalten erhält, bei kleinen (sm) sechs, bei mittleren (md) vier und bei großen (lg) drei der insgesamt zwölf Spalten. Somit werden je nach Auflösung eine bis vier Karten nebeneinander präsentiert. Hierzu sieht Bootstrap die nachfolgend verwendeten Klassen vor:

<div \*ngFor="let f of flights" class="col-xs-12 col-sm-6 col-md-4 col-lg-3">

<flight-card [item]="f" [selected]="basket[f.id]">

</flight-card>

</div>

Jede dieser Klassen, die mit dem Präfix col- eingeleitet werden, gibt für eine Auflösung die gewünschte Spaltenanzahl ein. Beispielsweise bedeutet col-md-4, dass eine Karte bei einem mittleren Gerät vier der zwölf Spalten erhält.

Hinweis Ende

## Komponenten mit Event Bindings

Dieser Abschnitt erweitert die hier gezeigte *FlightCardComponent* um ein Ereignis *selectedChange*. Dieses Ereignis soll den Parent informieren, wenn die Karte aus- bzw. abgewählt wird. Listing 6 zeigt, wie es verwendet werden soll.

<div \*ngFor="let f of flights">

<flug-card [item]="f"

[selected]="basket[f.id]"

(selectedChange)="basket[f.id] = $event">

</flug-card>

</div>

Listing 6: Nutzung einer Komponente mit Events

Man könnte sich solch eine Komponente als Funktion vorstellen, die Parameter übernimmt und über einen Callback Informationen veröffentlicht.

Die Metapher mit dem Chip passt hier noch besser. Dieser hat Ein- und Ausgänge, über den er mit seiner Umgebung verdrahtet wird. Die Ausgänge entsprechen den Events (Abbildung 9). Im hier betrachteten Fall fließt der Wert *selected* über einen Ausgang zurück in den Warenkorb.

Abbildung 9: Komponente mit Event-Bindings als Chip

### Implementierung der Komponente mit Event-Binding

Für das Event erhält die *FlightCardComponent* eine Eigenschaft *selectedChange*, welche mit *Output* zu dekorieren ist (Listing 7). Ihr Typ ist per Definition ein *EventEmitter*.

import { Component, EventEmitter, Input, Output } from '@angular/core';

import { Flight } from '../../entities/flight';

@Component({

selector: 'flight-card',

templateUrl: './flight-card.component.html'

})

export class FlightCardComponent {

@Input() item: Flight;

@Input() selected: boolean;

@Output() selectedChange = new EventEmitter<boolean>();

select() {

this.selected = true;

this.selectedChange.emit(this.selected);

}

unselect() {

this.selected = false;

this.selectedChange.emit(this.selected);

}

}

Listing 7: Komponente mit Event

Damit der *EventEmitter* den neuen Wert von *selected* über *$event* veröffentlichen kann, wird er mit Boolean typisiert.

### Komponente aufrufen

Nach dieser Erweiterung, kann der Aufruf der *FlightCardComponent* einen Event-Handler für *selectedChange* festlegen (Listing 8).

<div \*ngFor="let f of flights">

<flight-card [item]="f"

[selected]="basket[f.id]"

(selectedChange)="basket[f.id] = $event">

</flight-card>

</div>

Listing 8: Festlegen eines Event-Handlers

Die Anwendung sollte nun beim Aus- und Abwählen einer Karte den Warenkorb aktualisieren (Abbildung 10).

Abbildung 10: Der Warenkorb wird nun aktualisiert

## Komponenten mit Two-Way-Bindings

Die meisten Leser dürften es schon bemerkt haben: Die *Input*/*Output*-Kombination für *selected* erfüllt sämtliche in Abschnitt *Two-Way-Bindings* diskutierten Konventionen für die verkürzte Banana-in-a-Box-Schreibweise: Das Event setzt sich aus dem Namen der Property sowie aus dem Suffix Change zusammen und veröffentlicht den geänderten Wert via *$event*. Insofern spricht hier nichts gegen den Einsatz dieser komfortablen Grammatik (Listing 9).

<div \*ngFor="let f of flights">

<flight-card [item]="f"

[(selected)]="basket[f.id]">

</flight-card>

</div>

Listing 9: Verkürzte Schreibweise für Two-Way-Databinding

Hier wird auch der Nachteil dieser Abkürzung ersichtlich: Sie schreibt nach jeder Änderung den neuen Wert direkt in die Ausgangsvariable retour. Wollte man hingen bei einer Änderung eine Methode anstoßen, müsste man das stattdessen explizit mit einem Event-Binding erledigen.

# Life Cycle Hooks

Eine Komponente unterliegt einem bestimmten Lebenszyklus: Sie wird irgendwann erzeugt, erhält Daten über Property-Bindings und wird irgendwann auch wieder zerstört. Letzteres ist z. B. der Fall, wenn die Bedingung eines umgebenden *ngIf* nicht mehr erfüllt ist.

Angular-Anwendungen können auf diese Stationen im Leben einer Komponente reagieren, indem sie Lifecycle-Hooks implementieren.

## Ausgewählte Hooks

Angular bietet Hooks für verschiedene Zeitpunkte im Leben einer Komponente. Hier werden drei davon vorgestellt. Weitere Kapitel führen Anlassbezogen weitere ein.

Für jeden Life-Cycle-Hook definiert Angular einen Supertyp im Modul *@angular/core*, der eine Methode vorgibt (Tabelle 0‑1).

|  |  |  |
| --- | --- | --- |
| Supertyp | Methode | Beschreibung |
| OnInit | ngOnInit | Wird nach dem Initialisieren und somit nach dem ersten Ausführen der Property-Bindings aufgerufen. |
| OnChanges | ngOnChanges | Wird nach jedem Property-Binding aufgerufen. Der erste Aufruf erfolgt kurz vor dem Aufruf von OnInit. |
| OnDestroy | ngOnDestroy | Wird aufgerufen, bevor Angular eine Komponente zerstört. |

Tabelle 0‑1: Ausgewählte Live-Cycle-Hooks

Um nun Hooks zu nutzen, implementiert die gewünschte Komponente die jeweiligen Super-Typen und dessen Methoden (Listing 10).

@Component({

selector: 'my-component',   
 […]

})

export class Component implements OnChanges, OnInit {

@Input() someData;

ngOnInit() {

[…]

}

ngOnChanges() {

[…]

}

}

Listing 10: Nutzung von Life-Cycle-Hooks

## Experiment mit Life-Cycle-Hooks

Zum Vertrautmachen mit Life-Cycle-Hooks wird hier ein kleines Experiment beschrieben. Es erweitert die weiter oben eingeführte *FlightCardComponent* um die Hooks *OnInit* und *OnChange* (Listing 11).

import { Flight } from './../../entities/flight';

import { Component, Input, EventEmitter, Output, OnInit, OnChanges } from '@angular/core';

@Component({

selector: 'flight-card',

templateUrl: './flight-card.component.html'

})

export class FlightCardComponent implements OnInit, OnChanges {

@Input() item: Flight;

@Input() selected: boolean;

@Output() selectedChange = new EventEmitter<boolean>();

constructor() {

console.debug('ctor', this.item);

}

ngOnInit() {

console.debug('ngOnInit', this.item);

}

ngOnChanges(changes) {

console.debug('ngOnChanges', this.item);

if (changes.item) {

console.debug('ngOnChanges: item');

}

if (changes.selected) {

console.debug('ngOnChanges: selected');

}

}

select() {

this.selected = true;

this.selectedChange.next(this.selected);

}

deselect() {

this.selected = false;

this.selectedChange.next(this.selected);

}

}

Listing 11: FlightCardComponent mit Live-Cycle-Hooks

Die beiden von den Hooks vorgegebenen Methoden geben ihren Namen sowie den aktuellen Flug in der Eigenschaft *item* auf der Konsole aus. Die Methode *ngOnChange* prüft zusätzlich mit dem erhaltenen Parameter, welche Eigenschaften durch die letzte Property-Bindung aktualisiert wurden und notiert diese Erkenntnis auch auf der Konsole. Zum Vergleich gibt auch der Konstruktor den aktuellen Flug aus.

Lässt man nun diesen Code laufen, erhält man nach dem Flugsuchen die in Abbildung 11 gezeigte Ausgabe.

Abbildung 11: Debugging-Ausgaben bei Initialisierung von drei FlightCardComponent-Instanzen

Diese Ausgabe zeigt, dass Angular bei der ersten Datenbindung pro Komponente zuerst *ngOnChanges* und dann erst *ngOnInit* aufruft. Das entspricht auch den Informationen in Tabelle 0‑1. Außerdem mag es auf den ersten Blick verwundern, dass im Konstruktor item noch null ist. Das liegt daran, dass der Konstruktor das erste ist, das JavaScript für eine Klasse ausführt. Zu diesem Zeitpunkt hat Angular noch gar keine Gelegenheit gehabt, ein Databinding auszuführen. Möchte eine Komponente also auf gebundene Daten zugreifen, muss sie dazu *ngOnInit* oder *ngOnChange* nutzen.

Ändert man danach den Status einer Karte, erhält man nur den Aufruf von *ngOnChanges* (Abbildung 12).

Abbildung 12: Debugging-Ausgabe nach Auswahl einer FlightCardComponent-Instanz

## DateControl mit LifeCycleHooks

Als Ergänzung zum Experiment im letzten Abschnitt zeigt dieser Abschnitt eine Komponente, die von einem Life-Cycle-Hook abhängig ist. Es handelt sich dabei um eine einfache Datums-Komponente, die auf den Namen *DateComponent* hört (Abbildung 13).

Abbildung 13: Einfache Datums-Komponente

Um zu zeigen, dass die Komponente das Datum per Two-Way-Binding auch die gebundene Variable aktualisiert, gibt dieses Beispiel sie darunter zusätzlich per Datenbindung aus. Der Unterschied um eine Stunde ergibt sich durch die Nutzung der Mitteleuropäischen Winterzeit am verwendeten Rechner. Während die DateComponent diese Einstellung berücksichtigt, da es das Date-Objekt von JavaScript nutzt, ist das bei der Ausgabe nicht der Fall. Diese verwendet, wie die Endung Z andeutet, die Universalzeit, die auch in Greenwich zum Einsatz kommt.

Sie nimmt per Property Binding ein Datum in Form eines ISO-Strings entgegen und zerlegt es in seine Einzelteile. Diese bindet es an die präsentierten Eingabefelder. Da das Zerlegen des Datums immer dann zu erfolgen hat, wenn per Datenbindung ein neues Datum ankommt, kümmert sich darum der Life-Cycle-Hook *ngOnChange* (Listing 12).

import { Component, Input, OnInit, OnChanges, EventEmitter, Output } from '@angular/core';  
  
@Component({  
 selector: 'flight-date-component',  
 templateUrl: './date.component.html'  
})  
export class DateComponent implements OnInit, OnChanges {  
  
 @Input() date: string;  
 @Output() dateChange = new EventEmitter<string>();  
  
 day: number;  
 month: number;  
 year: number;  
 hour: number;  
 minute: number;  
  
 constructor() {  
 console.debug('date in constructor', this.date);  
 }  
  
 ngOnInit() {  
 console.debug('date in ngOnInit', this.date);  
 }  
  
 ngOnChanges(change) {  
 console.debug('date in ngOnChanges', this.date);  
 // if(change.date) { ... }  
  
 let date = new Date(this.date);  
 this.day = date.getDate();  
 this.month = date.getMonth() + 1;  
 this.year = date.getFullYear();  
 this.hour = date.getHours();  
 this.minute = date.getMinutes();  
 }  
  
 apply() {  
 let date = new Date(this.year, this.month - 1, this.day, this.hour, this.minute);  
 this.dateChange.next(date.toISOString());  
 }  
}

Listing 12: Die Datumskomponente zerlegt ein eingehendes Datum mit ngOnChanges

Hinweis Beginn

Um auf Änderungen an bestimmten Eigenschaften zu reagieren, kann als Alternative zu ngOnChanges auch mit TypeScript ein Setter eingeführt werden:

\_date: string;

@Input() set date(value: string) {

// Auf Wertänderung reagieren

this.\_date = value;

}

Hinweis Ende

Die Methode *apply* kommt zum Einsatz, wenn der Benutzer seine Eingaben bestätigt. Sie erstellt ein neues Datum aus den Einzelteilen und stößt mit einem davon abgeleiteten ISO-String das Event *dateChange* an.

Das Template dieser Komponente besteht lediglich aus Textfelder, die sich an die Teile des Datums binden. Außerdem weist es eine Schaltfläche, die die Methode *apply* aufruft, auf (Listing 13).

<form class="form-inline">  
 <input [(ngModel)]="day" name="day" maxlength="2" style="width:50px" class="form-control" >  
 .  
 <input [(ngModel)]="month" name="month" maxlength="2" style="width:50px" class="form-control">  
 .  
 <input [(ngModel)]="year" name="year" maxlength="4" style="width:70px" class="form-control">  
  
 <input [(ngModel)]="hour" name="hour" maxlength="2" style="width:50px" class="form-control">  
 :  
 <input [(ngModel)]="minute" name="minute" maxlength="2" style="width:50px" class="form-control">  
  
 <input type="button" value="Apply" (click)="apply()" class="btn btn-default">  
</form>

Listing 13: Die Datumskomponente zerlegt ein eingehendes Datum mit ngOnChanges

Damit die Angular von der Existenz der Komponente weiß, ist sie bei einem Modul zu registrieren. Auch hier kommt zur Vereinfachung das *AppModule* zum Einsatz (Listing 14).

@NgModule({

[...],

declarations: [

FlightSearchComponent,

FlightCardComponent,

DateComponent,

[...]

]

})

export class AppModule {

}

Listing 14: Komponente registrieren

Um die Komponente zu testen, erhält die *FlightSearchComponent* ein weiteres Feld *date*:

public date: string = (new Date()).toISOString();

Außerdem erhält ihr Template einen Verweis auf die *DateComponent* (Listing 15).

<div class="form-group">  
 <label>Date</label>  
 <flight-date-component [(date)]="date"></flight-date-component>  
 {{date}}  
</div>

Listing 15: Komponente einsetzen

# Zusammenfassung

Während bei Property-Bindings Daten im Komponentenbaum von oben nach unten fließen ist es bei Event-Bindings genau anders herum: Hier fließen die Daten von unten nach oben. Um Zyklen zu vermeiden, führt Angular diese Bindings in zwei Phasen durch. Nach jedem Event kommen die Event-Bindings zur Ausführung und danach kümmert es sich um die Property-Bindings, um die UI zu aktualisieren. Sogar Two-Way-Bindings sind streng genommen nur eine Kombination aus einem Property- und einem entgegengesetzten Event-Binding.

Eigene Komponenten können ebenfalls diese Bindings verwenden, um mit anderen Komponenten zu kommunizieren. Darüber hinaus benachrichtigt Angular jede Komponente mittels Life-Cycle-Hooks über bestimmte Ereignisse. Ein Beispiel dafür ist der Empfang initialer Daten oder neuer Daten über Property-Bindings.