6 Komponenten & Template-Syntax: Iteration I

»To be or not to be DOM. That's the question.«

Igor Minar
(Angular Lead Developer)

Nun, da unsere Projektumgebung vorbereitet ist, können wir beginnen, die ersten Schritte bei der Implementierung des BookMonkeys zu machen. Wir wollen in dieser Iteration die wichtigsten Grundlagen von Angular betrachten. Wir lernen zunächst das Prinzip der komponentenbasierten Entwicklung kennen und tauchen in die Template-Syntax von Angular ein. Zwei elementare Konzepte – die Property und Event Bindings – schauen wir uns dabei sehr ausführlich an.
Die Grundlagen von Angular sind umfangreich, deshalb müssen wir viel erläutern, bevor es mit der Implementierung losgeht. Aller Anfang ist schwer, aber haben Sie keine Angst – sobald Sie die Konzepte verinnerlicht haben, werden sie Ihnen den Entwickleralltag angenehmer machen!

6.1 Komponenten: Die Grundbausteine der Anwendung

Wir betrachten in diesem Abschnitt das Grundkonzept der Komponenten. Auf dem Weg lernen wir die verschiedenen Bestandteile der Template-Syntax kennen. Anschließend entwickeln wir mit der Listenansicht die erste Komponente für unsere Beispielanwendung.

6.1.1 Komponenten

Komponenten sind die Grundbausteine einer Angular-Anwendung. Jede Anwendung ist aus vielen verschiedenen Komponenten zusammengesetzt, die jeweils eine bestimmte Aufgabe erfüllen. Eine Komponente beschreibt somit immer einen kleinen Teil der Anwendung, z. B. eine Seite oder ein einzelnes UI-Element.

Hauptkomponente

Jede Anwendung besitzt mindestens eine Komponente, die Hauptkomponente (engl. *Root Component*). Alle weiteren Komponenten sind dieser Hauptkomponente untergeordnet. Eine Komponente hat außerdem einen Anzeigebereich, die *View*, in dem ein Template dargestellt wird. Das Template ist das »Gesicht« der Komponente, also der Bereich, den der Nutzer sieht.

Eine Komponente besitzt immer ein Template.

An eine Komponente wird üblicherweise Logik geknüpft, die die Interaktion mit dem Nutzer möglich macht.

Komponente: Klasse mit Decorator @Component()

Das Grundgerüst sieht wie folgt aus: Eine Komponente besteht aus einer TypeScript-Klasse und wird immer mit dem Decorator `@Component()` eingeleitet. Das Listing 6–1 zeigt den Grundaufbau einer Komponente.

Was ist ein Decorator?

Dekoratoren dienen der Angabe von Metainformationen zu einer Komponente. Der Einsatz von Metadaten fördert die Übersichtlichkeit im Code, da Konfiguration und Ablaufsteuerung sauber voneinander getrennt werden. Wer mit der Verwendung von Dekoratoren noch nicht vertraut ist, sollte sich den Abschnitt »Dekoratoren« auf Seite 42 durchlesen.

Listing 6–1 *Eine simple Komponente*

```
@Component({
  selector: 'my-component',
  template: '<h1>Hallo Angular!</h1>'
})
export class MyComponent { }
```

Metadaten

Selektor

Dem Decorator werden die *Metadaten* für die Komponente übergeben. Beispielsweise wird mit der Eigenschaft `template` das Template für die Komponente festgelegt. Im Property `selector` wird ein CSS-Selektor angegeben. Damit wird ein HTML-Element ausgewählt, an das die Komponente gebunden wird.

Was ist ein CSS-Selektor?

Mit CSS-Selektoren wählen wir Elemente aus dem DOM aus. Es handelt sich um dieselbe Syntax, die wir in CSS-Stylesheets verwenden, um Elementen einen Stil zuzuweisen. In Angular nutzen wir den Selektor unter anderem, um eine Komponente an eine Auswahl von Elementen zu binden. In Tabelle 6–1 sind die geläufigsten Selektoren aufgelistet. Selektoren können kombiniert werden, um die Auswahl weiter einzuschränken. Die Möglichkeiten sind sehr vielfältig, und wir nennen an dieser Stelle nur einige Beispiele:

- `div.active` – Containerelemente, die die CSS-Klasse `active` besitzen
- `input[type=text]` – Eingabefelder vom Typ `text`
- `li:nth-child(2)` – jedes zweite Listenelement innerhalb desselben Elternelements

Tab. 6–1 *Geläufige CSS-Selektoren*

Selektor	Beschreibung
`my-element`	Elemente mit dem Namen `my-element` Beispiel: `<my-element></my-element>`
`[myAttr]`	Elemente mit dem Attribut `myAttr` Beispiel: `<div myAttr="foo"></div>`
`[myAttr=bar]`	Elemente mit dem Attribut `myAttr` und Wert `bar` Beispiel: `<div myAttr="bar"></div>`
`.my-class`	Elemente mit der CSS-Klasse `my-class` Beispiel: `<div class="my-class"></div>`

Host-Element

Das Element stellt dann die Logik und das Template der Komponente bereit und wird deshalb als *Host-Element* bezeichnet. Wir betrachten noch einmal das Listing 6–1: Verwenden wir das DOM-Element `<my-component>` in unserem Template, so wird Angular den vorherigen Inhalt des Elements mit dem neuen Inhalt der Komponente ersetzen. Das Element `<my-component>` wird das Host-Element für diese Komponente, und es wird folgendes Markup erzeugt:

Listing 6–2 *Erzeugtes Markup für die Komponente* MyComponent

```
<my-component>
  <h1>Hallo Angular!</h1>
</my-component>
```

Wir können dieses Element an beliebiger Stelle in unseren Templates verwenden – es wird immer durch die zugehörige Komponente ersetzt. Auf diese Weise können wir Komponenten beliebig tief verschachteln, indem wir im Template einer Komponente das Host-Element einer an-

deren einsetzen usw. Diese Praxis schauen wir uns im nächsten Kapitel ab Seite 88 genauer an.

Komponenten sollten nur auf Elementnamen selektieren.

Es ist eine gute Praxis, stets nur Element*namen* zu verwenden, um Komponenten einzubinden. Das Prinzip der Komponente – Template und angeheftete Logik – kann durch ein eigenständiges Element am sinnvollsten abgebildet werden. In manchen Fällen ist es allerdings nicht zu vermeiden, Komponenten über Attribute oder CSS-Klassen an ein Element zu binden. Diesen Fall werden wir im Verlauf des Buchs auch noch kennenlernen.

Wenn wir auf die Attribute eines Elements selektieren wollen, sollten wir allerdings vorrangig *Attributdirektiven* einsetzen. Wie das funktioniert und wie wir eigene Direktiven implementieren können, schauen wir uns ab Seite 282 an.

Das Template einer Komponente

Eine Komponente ist immer mit einem Template verknüpft. Das Template ist der Teil der Komponente, den der Nutzer sieht und mit dem er interagieren kann. Für die Beschreibung wird meist HTML verwendet[1], denn wir wollen unsere Anwendung ja im Browser ausführen. Innerhalb der Templates wird allerdings eine Angular-spezifische Syntax eingesetzt, denn Komponenten können weit mehr, als nur statisches HTML darzustellen. Diese Syntax schauen wir uns im Verlauf dieses Kapitels noch genauer an.

Um eine Komponente mit einem Template zu verknüpfen, gibt es zwei Wege:

- **Component Inline Template:** Das Template wird als (mehrzeiliger) String im Quelltext der Komponente angegeben (`template`).
- **Template-URL:** Das Template liegt in einer eigenständigen HTML-Datei, die in der Komponente referenziert wird (`templateUrl`).

Eine Komponente besitzt genau ein Template.

In beiden Fällen verwenden wir die Metadaten des `@Component()`-Decorators, um die Infos anzugeben. Im Listing 6–3 sind beide Varianten zur Veranschaulichung aufgeführt. Es kann allerdings immer nur einer der beiden Wege verwendet werden, denn eine Komponente besitzt nur ein einziges Template. Das Template in eine externe Datei auszulagern ist besonders dann sinnvoll, wenn der Quelltext zu lang wird. Kurze Templates sollten inline definiert werden.

[1] Später im Kapitel zu NativeScript (ab Seite 431) werden wir einen Einsatzzweck ohne HTML kennenlernen.

Listing 6–3
Template einer Komponente definieren

```
@Component({

  // Als einzeiliger String mit Single Quotes
  template: '<h1>Hallo Angular!</h1>',

  // ODER: Als mehrzeiliger String mit Backticks
  template: `<h1>
          Hallo Angular!
  </h1>`,

  // ODER: Als Referenz zu einem HTML-Template
  templateUrl: './my-component.html',

  // [...]
})
export default class MyComponent { }
```

Inline-Templates mit der Angular CLI

Verwenden wir die Angular CLI, um eine Komponente zu erstellen, wird das Template standardmäßig in einer separaten Datei angelegt. Geben wir den Parameter `--inline-template` (Kurzform: `-it`) an, wird ein Inline-Template angelegt.

```
ng g component foo-bar -it
```

Bindings für die Kommunikation zwischen Komponente und Template

Template und Komponente sind eng miteinander verknüpft und können über klar definierte Wege miteinander kommunizieren. Der Informationsaustausch findet über sogenannte *Bindings* statt. Damit »fließen« die Daten von der Komponente ins Template und können dort dem Nutzer präsentiert werden. Umgekehrt können Ereignisse im Template abgefangen werden, um von der Komponente verarbeitet zu werden. Diese Kommunikation ist schematisch in Abbildung 6–1 dargestellt.

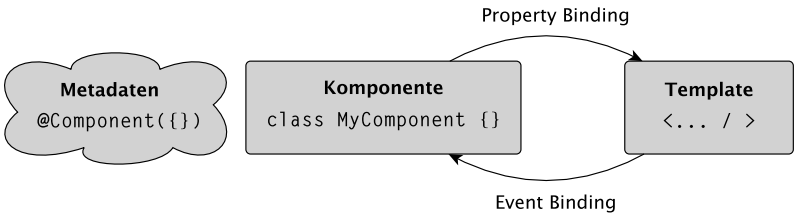


Abb. 6–1
Komponente, Template und Bindings im Zusammenspiel

Um diese Bindings zu steuern, nutzen wir die Template-Syntax von Angular, die wir gleich noch genauer betrachten. In den beiden folgenden Storys dieser Iteration gehen wir außerdem gezielt auf die verschiedenen Arten von Bindings ein.

Der Style einer Komponente

Um das Aussehen einer Komponente zu beeinflussen, werden Cascading Style Sheets (CSS) eingesetzt, wie wir sie allgemein aus der Webentwicklung kennen. Normalerweise verwendet man eine große, globale CSS-Datei, die alle gestalterischen Aspekte der Anwendung definiert. Das ist nicht schön, denn hier kann man leicht den Überblick verlieren, welche Selektoren wo genau aktiv sind oder womöglich gar nicht mehr benötigt werden. Außerdem widerspricht eine globale Style-Definition dem modularen Prinzip der Komponenten.

Stylesheets von Komponenten sind isoliert.

Angular zeigt hier einen neuen Weg auf und ordnet die Styles direkt den Komponenten zu. Diese direkte Verknüpfung von Styles und Komponenten sorgt dafür, dass die Styles einen begrenzten Gültigkeitsbereich haben und nur in ihrer jeweiligen Komponente gültig sind. Styles von zwei voneinander unabhängigen Komponenten können sich damit nicht gegenseitig beeinflussen, sind bedeutend übersichtlicher und liegen immer direkt am »Ort des Geschehens« vor.

Ein Blick ins Innere: View Encapsulation

Styles werden einer Komponente zugeordnet und wirken damit auch nur auf die Inhalte dieser Komponente. Die Technik dahinter nennt sich *View Encapsulation* und isoliert den Gültigkeitsbereich eines Anzeigebereichs von anderen. Jedes DOM-Element in einer Komponente erhält automatisch ein zusätzliches Attribut mit einem zufälligen Bezeichner, siehe Screenshot. Die vom Entwickler festgelegten Styles werden abgeändert, sodass sie nur für dieses Attribut wirken. So funktioniert der Style nur in der Komponente, in der er deklariert wurde. Es gibt noch andere Strategien der View Encapsulation, auf die wir aber hier nicht eingehen wollen.

Angular generiert zufällige Attribute für die View Encapsulation.

Die Styles werden ebenfalls in den Metadaten einer Komponente angegeben. Dafür sind zwei Wege möglich, die wir auch schon von den Templates kennen:

- **Component Inline Styles:** Die Styles werden direkt in der Komponente definiert (`styles`).
- **Style-URL:** Es wird eine CSS-Datei mit Style-Definitionen eingebunden (`styleUrls`).

Im Listing 6–4 werden beide Wege gezeigt. Wichtig ist, dass die Styles und Dateien jeweils als Arrays angelegt werden. Kurze Style-Definitionen können inline eingebunden werden. Für längere Abschnitte empfiehlt es sich, eine eigene Datei anzulegen und der Komponente zuzuweisen.

Die herkömmlichen Varianten sind natürlich trotzdem weiter möglich: Wir können globale CSS-Dateien einbinden[2] oder Styles im Templates direkt für einzelne Elemente definieren.

Listing 6–4
Style-Definitionen in Komponenten

```
@Component({
  styles: [
    'h2 { color:blue }',
    'h1 { font-size: 3em }'
  ],
  // ODER
  styleUrls: ['./stylesheet1.css','./stylesheet2.css'],
  // [...]
})
export default class MyComponent { }
```

Inline-Styles mit der Angular CLI

Wenn wir mit der Angular CLI eine Komponente generieren, wird das Stylesheet standardmäßig in einer separaten Datei angelegt. Mit dem Parameter `--inline-style` (Kurzform: `-is`) werden Component Inline Styles verwendet.

```
ng g component foo-bar -is
```

6.1.2 Komponenten in der Anwendung verwenden

Rule of One

Eine Komponente wird immer in einer eigenen TypeScript-Datei notiert. Dahinter steht das *Rule of One*: Eine Datei beinhaltet immer genau einen Bestandteil und nicht mehr. Dazu kommen meist ein separates Template, eine Style-Datei und eine Testspezifikation. Diese vier Dateien sollten wir immer gemeinsam in einem eigenen Ordner unter-

[2] Diesen Weg haben wir gewählt, um das Style-Framework Semantic UI einzubinden, siehe Seite 59.

bringen. So wissen wir sofort, welche Dateien zu der Komponente gehören.

Eine Komponente besitzt einen Selektor und wird automatisch an die DOM-Elemente gebunden, die auf diesen Selektor matchen. Das jeweilige Element wird das Host-Element der Komponente. Das Prinzip haben wir einige Seiten zuvor schon beleuchtet.

Komponenten im AppModule *registrieren*

Damit dieser Mechanismus funktioniert, muss Angular die Komponente allerdings erst kennenlernen. Die reine Existenz einer Komponentendatei reicht nicht aus. Stattdessen müssen wir alle Komponenten der Anwendung im zentralen `AppModule` registrieren.

Dazu dient die Eigenschaft `declarations` im Decorator `@NgModule()`. Hier werden alle Komponenten[3] notiert, die zur Anwendung gehören. Damit wir die Typen dort verwenden können, müssen wir alle Komponenten importieren.

Listing 6–5 *Alle Komponenten müssen im* AppModule *deklariert werden.*

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

import { AppComponent } from './app.component';
import { FooComponent } from './foo/foo.component';
import { BarComponent } from './bar/bar.component';

@NgModule({
  declarations: [
    AppComponent,
    FooComponent,
    BarComponent
  ],
  // ...
})
export class AppModule { }
```

6.1.3 Template-Syntax

Nachdem wir die Grundlagen von Komponenten kennengelernt haben, tauchen wir nun in die Notation von Templates ein.

Angular erweitert die gewohnte Syntax von HTML mit einer Reihe von leichtgewichtigen Ausdrücken. Damit können wir dynamische Features direkt in den HTML-Templates nutzen: Ausgabe von Daten, Reaktion auf Ereignisse und das Zusammenspiel von mehreren Komponenten mit Bindings.

[3] ... und Pipes und Direktiven, aber dazu kommen wir später!

Micro-Syntax

Die hier vorgestellte Schreibweise wird als *Micro-Syntax* bezeichnet, denn für jeden dieser Ausdrücke existiert auch eine Langform, die *kanonische Form*. Ob die Kurzschreibweise oder die lange Variante gewählt wird, ist reine Geschmackssache. In diesem Abschnitt verzichten wir allerdings bewusst auf die kanonische Form, denn sie wird im Alltag normalerweise nicht eingesetzt.

Wir stellen in diesem Abschnitt die Einzelheiten der Template-Syntax zunächst als Übersicht vor. Im Verlauf des Buchs gehen wir auf die einzelnen Konzepte noch gezielter ein.

{{ Interpolation }}

Bereits in AngularJS 1.x konnten wir Daten mit zwei geschweiften Klammern in ein HTML-Template einbinden. Dieses Konzept bleibt in Angular erhalten und wird mit dem etwas sperrigen Begriff *Interpolation* bezeichnet.

Template-Ausdruck

Zwischen den Klammern wird ein sogenannter *Template-Ausdruck* angegeben. Im einfachsten Fall ist das der Name einer Eigenschaft aus der aktuellen Komponentenklasse. Ausdrücke können aber auch komplexer sein und z. B. Arithmetik enthalten. Vor der Ausgabe werden die Ausdrücke ausgewertet und der Rückgabewert wird schließlich im Template angezeigt. Besonders interessant dabei: Ändern sich die Daten im Hintergrund, wird die Anzeige stets automatisch aktualisiert!

Die Daten werden bei der Interpolation automatisch aktualisiert.

```
{{ name }} (Property der Komponente)
{{ 'foobar' }} (String-Literal)
{{ myNumber + 1 }} (Arithmetik)
```

Die Werte `null` und `undefined` werden immer als leerer String ausgegeben. Doch wie sieht es bei komplexeren Datenstrukturen aus?

Der Safe-Navigation-Operator ?

Stellen wir uns vor, dass wir die Felder eines Objekts in unserem Template anzeigen möchten. Wenn das Objekt `undefined` oder `null` ist, erhalten wir einen Fehler zur Laufzeit der Anwendung, weil es nicht möglich ist, auf eine Eigenschaft eines nicht vorhandenen Objekts zuzugreifen.

An dieser Stelle hilft der *Safe-Navigation-Operator* aus. Er ist ein nützliches Instrument, um vor der Verwendung zu prüfen, ob ein Objekt vorhanden ist oder nicht. Falls nicht, wird der Ausdruck zum Binden der Daten gar nicht erst ausgewertet.

```
<p>{{ person?.hobbies }}</p>
```

Komplexere Objektbäume

Hier prüfen wir, ob das Objekt `person` überhaupt existiert. Wenn ja, werden die Hobbys der Person ausgegeben. Der `?`-Operator funktioniert auch in komplexeren Objektbäumen:

```
<p>{{ person?.hobbies?.coding?.angular }}</p>
```

Der Operator bietet sich vor allem dann an, wenn Daten optional sind oder erst zu einem späteren Zeitpunkt gebunden werden, z. B. bei einem asynchronen HTTP-Request.

[Property Bindings]

Property Bindings werden automatisch aktualisiert.

Mit Property Bindings werden Daten von außen an ein DOM-Element übermittelt. Angegeben wird ein Template-Ausdruck, also wie bei der Interpolation z. B. ein Property der Komponente oder ein Literal. Ist das Element ein Host-Element einer Komponente, können wir die Daten innerhalb dieser Komponente auslesen und verarbeiten. Property Bindings werden ebenfalls automatisch aktualisiert, wenn sich die Daten ändern. Wir müssen uns also nicht darum kümmern, Änderungen an den Daten manuell mitzuteilen.

Das folgende Beispiel zeigt ein Property Binding im praktischen Einsatz. Wir setzen damit das Property `href` des Link-Elements auf den Wert der Komponenteneigenschaft `myUrl`.

```
<a [href]="myUrl">My Link</a>
```

Property Bindings werden im Abschnitt ab Seite 88 ausführlich behandelt.

Eselsbrücke

Um in JavaScript auf eine Eigenschaft eines Objekts zuzugreifen, verwenden wir ebenfalls eckige Klammern: `obj[property]`.

(Event Bindings)

Gegenstück zu Property Bindings

Event Bindings bieten die Möglichkeit, auf Ereignisse zu reagieren, die im Template einer Komponente eintreten. Solche Ereignisse können nativ von einem DOM-Element stammen (z. B. ein Klick) oder in einer eingebundenen Komponente getriggert werden. Event Bindings sind also das Gegenstück zu Property Bindings, denn die Daten fließen aus dem Template in die Komponentenklasse. Im folgenden Beispiel wird ein `click`-Event ausgelöst, wenn der Nutzer den Button anklickt. Das Ereignis wird mit der Methode `myClickHandler()` abgefangen und behandelt.

```
<button (click)="myClickHandler()">Click me</button>
```

Wie wir Events in Komponenten auslösen und welche eingebauten Events abonniert werden können, schauen wir uns ausführlich ab Seite 101 an.

Eselsbrücke

In JavaScript verwenden wir runde Klammern für Funktionen: `function()`. Vor diesem Hintergrund lässt sich die Syntax für Event Bindings auch leicht merken, denn wir führen eine Funktion aus, nachdem ein Ereignis aufgetreten ist.

[(Two-Way Bindings)]

Mit Property Bindings haben wir »schreibende« und mit den Event Bindings »lesende« Operationen kennengelernt. Beide Varianten sind unidirektional, die Daten »fließen« also nur in eine Richtung.

Rückblick zu AngularJS 1.x

In AngularJS fand der Informationsaustausch standardmäßig in beide Richtungen (Two-Way Binding) statt. Dieses Prinzip wurde mit Angular ab Version 2 gebrochen und einzelne Bindings für beide Richtungen wurden eingeführt.

Anwendungen mit AngularJS 1.x hatten die unangenehme Eigenschaft, mit zunehmender Größe immer schwerfälliger zu werden. Das lag vor allem daran, dass alle aktiven Two-Way Bindings kontinuierliche Überprüfungen verlangen. Erst mit AngularJS 1.3 haben die lang ersehnten einmaligen (One-Time) Databindings Einzug gehalten. Zum einen wird so die Syntax viel klarer, zum anderen ist es so technisch viel einfacher, eine gute Performance der Anwendung zu garantieren.

Die Direktive ngModel

Vor allem bei der Formularverarbeitung ist es allerdings praktisch, wenn sich die Daten in beide Richtungen aktualisieren, denn Änderungen können sowohl von der Komponente an ein Formularfeld gehen als auch vom Formularfeld an die Komponente übermittelt werden. Mit der Direktive `ngModel` können wir deshalb ein Property an ein Formularfeld binden. Ändert sich der Inhalt des Propertys, wird das Formular automatisch aktualisiert. Wird der Inhalt des Formulars geändert, wird der neue Wert automatisch in das Property geschrieben. Die Syntax ist sehr einfach herzuleiten, denn es werden lediglich Property Bindings und Event Bindings miteinander kombiniert:

```
<input [(ngModel)]="myProperty" type="text">
```

Mit den eckigen Klammern binden wir einen Wert aus der Komponente an das `<input>`-Element. Über die runden Klammern verarbeiten wir die Änderungen, die aus dem Element kommen.

In der Iteration IV ab Seite 207 schauen wir uns die Formularverarbeitung noch sehr ausführlich an. Dort werden wir Two-Way Bindings verwenden, um Formulareingaben zwischen Template und Komponente auszutauschen.

Eselsbrücke

Für diese Eselsbrücke ist etwas Phantasie nötig. Die Kombination von runden und eckigen Klammern (`[()]`) sieht ein wenig wie eine Banane in einer Kiste aus (»Banana Box«). Alternativ funktioniert auch eine Eselsbrücke aus dem Ballsport: *Das Runde muss ins Eckige.*

#Elementreferenzen

In einem Template können wir einzelne HTML-Elemente mit Namen versehen und diese Namen an anderer Stelle verwenden, um auf das Element zuzugreifen. Damit können wir aus dem Template heraus die Eigenschaften eines Elements verwenden, ohne den Umweg über die Komponentenklasse zu gehen. Solche Elementreferenzen werden mit einem Rautensymbol # notiert.

```
<input #id type="text" value="Angular">
{{ id.value }}
```

Input-Felder haben beispielsweise immer ein Property `value`, in dem der aktuelle Wert hinterlegt ist. Das Beispiel `{{ id.value }}` macht deutlich, dass die lokale Referenz tatsächlich auf das HTML-Element zeigt und nicht nur auf dessen Wert. Beim Start wird der Text `Angular` neben dem Formularfeld ausgegeben. Geben wir einen neuen Wert in das Input-Feld ein, aktualisiert sich der interpolierte Text allerdings nicht. Das ist kein Fehler, sondern ein erwünschtes Verhalten von Angular.[4]

*Strukturdirektiven

Wir haben gelernt, dass wir mit Property und Event Bindings den DOM und Komponenten beeinflussen können. Damit können wir zwar zwischen existierenden Elementen kommunizieren, allerdings fehlt uns noch ein Konstrukt, um Knoten im DOM-Baum neu zu strukturieren.

[4] Angular geht sparsam mit den Ressourcen um, und führt die sogenannte Change Detection erst dann aus, wenn sie durch ein Event ausgelöst wird. Das kann durch ein Event Binding geschehen oder durch ein Two-Way Binding.

So möchten wir zum Beispiel ein Element nur unter einer Bedingung hinzufügen oder sie mehrfach wiederholen können.

Hierfür gibt es spezielle HTML-Attribute, die zusammen mit dem Stern-Zeichen `*` verwendet werden. Sie werden *Strukturdirektiven* (Structural Directives) genannt. Diese Bezeichnung rührt daher, dass sie DOM-Elemente hinzufügen oder entfernen und somit die Struktur des Dokuments verändern. Das entsprechende DOM-Element bekommt im Zuge dessen neue Logik zugeordnet. Wir haben das Prinzip schon bei den Komponenten kennengelernt. Komponenten besitzen allerdings immer ein Template und können nur einmal auf ein Element angewendet werden. Bei Direktiven besteht diese Beschränkung nicht, und wir können ein Element mit mehreren Direktiven versehen.

Strukturdirektiven verändern die Struktur des DOM-Baums.

Die bekanntesten Vertreter der Strukturdirektiven sind `ngIf`, `ngFor` und `ngSwitch`. Schauen wir uns die wichtigsten eingebauten Direktiven einmal an:

Die Direktive `ngIf` fügt das betroffene Element nur dann in den DOM-Baum ein, wenn der angegebene Ausdruck wahr ist. Sie kann z. B. eingesetzt werden, um ein Element mit einem Warnhinweis nur dann anzuzeigen, wenn es nötig ist.

ngIf: Bedingungen zum Anzeigen von Elementen

```
<div *ngIf="true">Lorem ipsum</div>
```

Listing 6–6
ngIf *verwenden*

Ab Angular in der Version 4 verfügt die Direktive zusätzlich über einen optionalen Else-Zweig. Mehr dazu erfahren Sie im Kapitel »Wissenswertes« ab Seite 516.

Mit der Direktive `ngFor` können wir ein Array durchlaufen und für jedes iterierte Array-Element ein DOM-Element erzeugen. Damit können wir z. B. Tabellenzeilen oder Listen in HTML ausgeben. Im Beispiel durchlaufen wir das Array `names` aus der aktuellen Komponente. Das aktuelle Element ist bei jedem Durchlauf in der lokalen Variable `name` hinterlegt. Dieser Name ist frei wählbar.

ngFor: Listen durchlaufen

```
<ul>
  <li *ngFor="let name of names">{{ name }}</li>
</ul>
```

Listing 6–7
ngFor *verwenden*

Ist das Array `names` beispielsweise eine Liste mit vier Namen, so erzeugt Angular das folgende Markup:

```
<ul>
  <li>Gregor</li>
  <li>Ferdinand</li>
  <li>Danny</li>
  <li>Johannes</li>
</ul>
```

Listing 6–8
Erzeugtes Markup für Listing 6–7

Hilfsvariablen für ngFor

`ngFor` stellt eine Reihe von Hilfsvariablen zur Verfügung, die wir beim Durchlaufen der Liste verwenden können. Sie beziehen sich jeweils auf das aktuelle Element:

- `index`: Index des aktuellen Elements `0...n`
- `first`: wahr, wenn es das *erste* Element der Liste ist
- `last`: wahr, wenn es das *letzte* Element der Liste ist
- `even`: wahr, wenn der Index *gerade* ist
- `odd`: wahr, wenn der Index *ungerade* ist

Diese Flags können wir dazu verwenden, um z. B. eine gestreifte Tabelle zu erzeugen oder nach jedem außer dem letzten Element ein Komma auszugeben. Die Variablen müssen vor der Verwendung jeweils auf lokale Variablen gemappt werden, z. B. `index as i`.

Das folgende Beispiel verwendet `index`, um vor jedem Namen eine fortlaufende Ziffer anzuzeigen. Die lokale Variable `i` wird vor der Anzeige jeweils um 1 erhöht, weil die Index-Zählung natürlich bei 0 beginnt.

Listing 6–9 ngFor *mit Hilfsvariablen*

```
<ul>
  <li *ngFor="let name of names; index as i">
    {{ i + 1 }}. {{ name }}
  </li>
</ul>
```

Das Beispiel erzeugt das folgende HTML:

Listing 6–10 *Erzeugtes Markup für Listing 6–9*

```
<ul>
  <li>1. Gregor</li>
  <li>2. Ferdinand</li>
  <li>3. Danny</li>
  <li>4. Johannes</li>
</ul>
```

ngSwitch*: Fallunterscheidungen*

Eine weitere wichtige Direktive ist `ngSwitch`. Damit lassen sich im Template Verzweigungen realisieren, wie sie sonst mit switch/case-Anweisungen möglich sind. `ngSwitch` wird immer zusammen mit zwei weiteren Direktiven eingesetzt: `ngSwitchCase` und `ngSwitchDefault`. Damit werden die Zweige für die Fallunterscheidung markiert. Die Direktiven sorgen dafür, dass immer der Zweig angezeigt wird, der dem Eingabewert entspricht.

Im folgenden Beispiel wird innerhalb des `<div>`-Elements über die Komponenteneigenschaft `angularVersion` entschieden. Je nachdem, ob der Wert `1`, `2` oder `4` ist, wird das zugehörige `<span>`-Element in den

DOM-Baum eingefügt. Wenn keiner der Fälle eintritt, wird das Default-Element ausgewählt.

Das Schlüsselwort `ngSwitch` wird mit eckigen Klammern angegeben. Das kommt daher, dass `ngSwitch` tatsächlich eine Attributdirektive ist. Den Unterschied schauen wir uns gleich an.

Listing 6–11
Einsatz von ngSwitch

```
<div [ngSwitch]="angularVersion">
  <span *ngSwitchCase="1">AngularJS</span>
  <span *ngSwitchCase="2">Angular</span>
  <span *ngSwitchCase="4">Angular</span>
  <span *ngSwitchDefault>Angular</span>
</div>
```

Wir werden uns in der Iteration V ab Seite 259 noch genauer mit Direktiven auseinandersetzen.

[Attributdirektiven]

Eine weitere Form der Direktiven lernen wir mit den *Attributdirektiven* (Attribute Directives) kennen. Während Strukturdirektiven den Aufbau des DOM-Baums verändern, wirken sich Attributdirektiven nur auf das Element selbst aus. Alle Direktiven haben gemeinsam, dass sie einem DOM-Element Logik hinzufügen.

Attributdirektiven wirken nur auf das jeweilige Element.

In ihrer Verwendung unterscheiden sich Attributdirektiven nicht von Property Bindings. Sie werden meist mit eckigen Klammern angegeben und erhalten einen Ausdruck. Der einzige Unterschied ist, dass mit Attributdirektiven außerdem Logik an das Element geheftet wird.

Bekannte Vertreter sind `ngSwitch`, `ngModel`, `ngClass` und `ngStyle`. Die ersten beiden haben wir in diesem Kapitel bereits kennengelernt. Auf `ngClass` und `ngStyle` gehen wir schließlich auf den Seiten 93 und 94 noch gezielter ein, wenn wir über Class und Style Bindings sprechen.

Der Pipe-Operator |

Pipes werden genutzt, um Daten für die Anzeige zu transformieren. Das Konzept entspricht den Filtern aus AngularJS 1.x. Pipes nehmen Eingabeargumente entgegen und liefern das transformierte Ergebnis zurück. In einem Template-Ausdruck werden sie durch das Symbol | (das »Pipe-Zeichen«) eingeleitet:

```
<p>{{ name | uppercase }}</p>
```

Pipes verketten

Pipes können auch aneinandergehängt werden, um mehrere Transformationen durchzuführen:

```
<p>{{ name | uppercase | lowercase }}</p>
```

In der Iteration V ab Seite 259 widmen wir uns den Pipes noch sehr ausführlich. Dort lernen wir auch, wie wir eigene Pipes implementieren können.

Was hat das mit HTML zu tun?

Obwohl die vielen verschiedenen Klammern und Sonderzeichen zunächst ungewöhnlich sind, handelt es sich um syntaktisch gültiges HTML. Dazu ziehen wir die HTML-Spezifikation des W3C zurate. Hier heißt es:

> »Attribute names must consist of one or more characters other than the space characters, U+0000, NULL, «, ', >, /, =, the control characters, and any characters that are not defined by Unicode.«[5]

Auch mit der Angular-Syntax schreiben wir also valides HTML. Trotzdem kennt der Browser natürlich keine Event oder Property Bindings. Deshalb werden all diese Ausdrücke von Angular automatisch in »richtiges« HTML und JavaScript umgewandelt, mit dem jeder Browser umgehen kann.

Zusammenfassung

Die Template-Syntax von Angular ist ein simples, aber mächtiges Werkzeug, um ein dynamisches Zusammenspiel von Komponente und Template zu erreichen. Jedes Verfahren verfügt über eine eigene Schreibweise. Alle Bestandteile der Template-Syntax sind noch einmal überblicksweise in der Tabelle 6–2 dargestellt.

Auf den ersten Blick erscheint die neue Template-Syntax womöglich komplex und umständlich. Aber glauben Sie uns: Nachdem wir alle Konzepte in der Praxis behandelt haben, werden Sie die Notationen ohne Probleme anwenden können.

[5] https://ng-buch.de/x/31 – W3C: HTML 5 Syntax

Zeichen	Bezeichnung	Funktion
{{}}	Interpolation	Daten im Template anzeigen
[]	Property Binding	Eigenschaften eines DOM-Elements setzen
()	Event Binding	Ereignisse abfangen und behandeln
[()]	Two-Way Binding	Eigenschaften lesen und Ereignisse verarbeiten
#	Elementreferenzen	Direktzugriff auf ein DOM-Element
*	Strukturdirektiven	Direktiven, die den DOM-Baum manipulieren
|	Pipe-Operator	Transformation von Daten vor dem Anzeigen

Tab. 6–2
Template-Syntax: Übersicht

6.1.4 Den BookMonkey erstellen

Story – Listenansicht

Als Leser möchte ich einen Überblick über alle vorhandenen Bücher erhalten, um mir den nächsten Lesetitel heraussuchen zu können.

Jedes Projekt braucht einen einfachen Anfang. Es bietet sich an, für den *BookMonkey* im ersten Schritt eine einfache Listenansicht für unsere Bücher zu implementieren. Weitere Funktionen werden später folgen.

- Ein Buch soll durch ein Vorschaubild dargestellt werden.
- Es sollen sowohl der Titel, der Untertitel als auch die Autoren des Buchs dargestellt werden.

Die Entität »Buch«

Wir müssen zunächst überlegen, wie die Entität *Buch* in unserer Anwendung repräsentiert werden kann. Dazu sammeln wir alle Eigenschaften, die ein Buch besitzen kann:

- `isbn`: Internationale Standardbuchnummer (ISBN)
- `title`: Buchtitel
- `description`: Beschreibung des Buchs
- `authors`: Liste der Autoren
- `subtitle`: Untertitel
- `rating`: Bewertung
- `published`: Datum der Veröffentlichung
- `thumbnails`: Liste von Bildern

Die Entität »Thumbnail«

Die ISBN ist unser Primärschlüssel, das heißt, ein Buch ist durch seine ISBN eindeutig identifizierbar. Einem Buch können über die Eigen-

schaft `thumbnails` mehrere Bilder zugeordnet sein. Ein solches Bild besitzt die folgenden Eigenschaften:

- `url`: URL zur Bilddatei
- `title`: Bildtitel

Model-Klassen für Buch und Thumbnail

Es bietet sich an, für jede der beiden Entitäten eine eigene TypeScript-Klasse zu erstellen, aus der sich dann Buch- und Thumbnail-Objekte instanziieren lassen. Diese Model-Klassen sollen in einem gemeinsamen Ordner 🗀`src/app/shared` untergebracht werden. Nachdem wir den Ordner angelegt haben, verwenden wir die Angular CLI, um die Klasse zu generieren. Als Erstes erstellen wir die Klasse `Thumbnail`.

Listing 6–12 *Klasse* Thumbnail *anlegen mit der Angular CLI*

```
$ ng g class shared/thumbnail
```

Der Befehl verlangt als Argument den Namen und Pfad der neuen Klasse und legt im Ordner 🗀`shared` die TypeScript-Datei für die Klasse an. Das Präfix 🗀`src/app` müssen wir dabei nicht angeben, auch wenn wir uns im Hauptverzeichnis der Anwendung befinden. Die neue Klasse beleben wir mit dem Modell unseres Thumbnails, wie es das Listing 6–13 zeigt. Die Klasse besteht nur aus einem Konstruktor, dem alle Werte als Argumente übergeben werden. Das Schlüsselwort `public` erzeugt dabei in der Klasse automatisch Eigenschaften (Propertys) mit demselben Namen. Damit stellen wir schnell und einfach sicher, dass ein Thumbnail immer garantiert alle diese Eigenschaften besitzt. Die Angabe `export` sorgt dafür, dass wir die Klasse an anderer Stelle wieder importieren können.

Listing 6–13 *Klasse* Thumbnail (thumbnail.ts)

```
export class Thumbnail {
  constructor(
    public url: string,
    public title: string
  ) { }
}
```

Als Nächstes erstellen wir die Klasse `Book` im Pfad `shared/book.ts`.

Listing 6–14 *Klasse* Book *anlegen mit der Angular CLI*

```
$ ng g class shared/book
```

Optionale Argumente im Konstruktor

Auch hier übergeben wir alle Werte des Buchs an den Konstruktor. Bei einem Buch müssen allerdings nicht immer alle Eigenschaften gesetzt sein. Deshalb wird das Fragezeichen im Namen dazu verwendet, den Compiler auf einen optionalen Parameter hinzuweisen. Fehlt einer der normalen Parameter beim Aufruf des Konstruktors, so schlägt das Kompilieren fehl. Bei den optionalen Parametern ist dies hingegen nicht der Fall.

Listing 6–15
Klasse Book (book.ts)

```
import { Thumbnail } from './thumbnail';
export { Thumbnail } from './thumbnail';

export class Book {
  constructor(
    public isbn: string,
    public title: string,
    public authors: string[],
    public published: Date,
    public subtitle?: string,
    public rating?: number,
    public thumbnails?: Thumbnail[],
    public description?: string
  ) { }
}
```

Die Eigenschaft `thumbnails` ist eine Liste von `Thumbnail`-Objekten. Damit wir diesen Typ hier verwenden können, müssen wir die Klasse importieren (Zeile 1).

Wir wollen `Book` später in anderen Teilen der Anwendung einsetzen und dort auch ein Array von Thumbnails übergeben. Dazu brauchen wir in jedem Fall beide Klassen und müssten immer zwei Dateien importieren, denn `Book` tritt stets in Begleitung von `Thumbnail` auf. Dafür gibt es eine schönere Lösung: Mit dem Export-Befehl in Zeile 2 exportieren wir hier auch die Klasse `Thumbnail`. Später können wir dann beide Klassen `Book` und `Thumbnail` aus nur einer Datei importieren, z. B. in einer unserer Komponenten:

Klassen importieren und exportieren

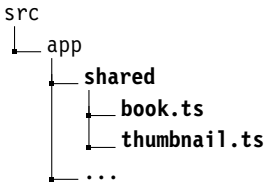
Listing 6–16
Import von Book *und* Thumbnail *aus* book.ts

```
import { Book, Thumbnail } from '../shared/book';
```

import und export

Die Schlüsselwörter `import` und `export` sind die Grundlage für ECMAScript-2015-Module. Wir werden unseren Quellcode ausschließlich in diesem Modulformat definieren.

Wir haben zwei neue Klassen angelegt, und unser Projekt sollte jetzt die folgende Ordnerstruktur besitzen:



Jetzt, da die Grundlage für die Bücher geschaffen ist, wollen wir unsere Buchliste angehen. Die Anwendung verfügt bereits über eine Root-Komponente. Zusätzlich dazu wollen wir nun eine weitere Komponente anlegen, die nur die Buchliste beinhaltet. Wir wählen dafür den Namen `BookListComponent` und führen das folgende Kommando mit der Angular CLI aus, um die Komponente zu erzeugen (ausgehend vom Hauptverzeichnis der Anwendung):

Listing 6–17 *Komponente* BookListComponent *mit der Angular CLI anlegen*

```
$ ng g component book-list
```

Die Angular CLI legt für jede Komponente automatisch einen neuen Ordner an, in dem die Komponentenklasse, die zugehörige Testspezifikation und das Template untergebracht sind. Zu dem Klassennamen wird automatisch das Suffix `Component` hinzugefügt! Diese Konvention entspricht dem Angular-Styleguide.[6]

Wir erhalten die folgende Ordnerstruktur:

```
src
└── app
    ├── book-list
    │   ├── book-list.component.html
    │   ├── book-list.component.spec.ts
    │   └── book-list.component.ts
    ├── shared
    │   └── ...
    └── ...
```

Die Angular CLI hält die Namenskonvention ein.

Es fällt positiv auf, dass sich die Angular CLI an die Namenskonventionen aus dem Styleguide hält (siehe Kasten). Die Klasse heißt nun `BookListComponent`, der Ordnername lautet 🗀`book-list`. Der eingegebene Name wurde also automatisch umgewandelt.

Namenskonventionen in Angular – kebap-case vs. CamelCase

In Angular wird eine strenge Namenskonvention verfolgt. Es werden grundsätzlich zwei verschiedene Formate genutzt:

- **kebap-case**: Diese Konvention wird für Dateinamen und Selektoren genutzt.
- **CamelCase**: Die CamelCase-Konvention wird für die Namensgebung von Klassen und Interfaces verwendet.

Entsprechend der Zielsetzung der ersten Iteration soll die Komponente eine Listenansicht von Büchern darstellen. Hierzu passen wir zunächst das generierte Template an. Wir legen das Grundgerüst für die Liste an

[6] https://ng-buch.de/x/32 – Angular Docs: Style Guide

und greifen dafür auf die Elemente des Style-Frameworks Semantic UI zurück.

ngFor für die Buchliste

Schließlich verwenden wir die Direktive `ngFor`, um durch die Liste der Bücher zu iterieren. `ngFor` wiederholt in unserem Fall das Element `<a class="item">` und dessen Inhalte für jedes Buch. In jedem dieser Blöcke werden der Titel, sofern vorhanden der Untertitel sowie die Autoren und die ISBN des Buchs ausgegeben. Wir verwenden die Interpolation, um Propertys der Komponentenklasse im Template auszugeben.

Autoren anzeigen mit ngFor

Die Liste der Autoren soll kommasepariert angezeigt werden. Wir verwenden hier ebenfalls `ngFor` und durchlaufen die Liste der Autoren. Um nach dem letzten Element kein Komma anzuzeigen, machen wir uns die Hilfsvariablen der Direktive zunutze: Die Variable `last` ist *true*, sobald die Schleife das letzte Element des Arrays erreicht hat. Wir können diesen Wert in eine lokale Variable `l` speichern und damit feststellen, wann der letzte Name ausgegeben wurde. Das Komma platzieren wir in einem `<span>`-Element, das wir mit der Direktive `ngIf` einblenden oder ausblenden, je nachdem, ob der letzte Name erreicht wurde oder nicht. Solange der letzte Wert nicht erreicht wurde, wird also hinter jedem Autor ein Komma und ein Leerzeichen angefügt.

Thumbnails anzeigen

Außerdem wird das erste Bild aus der Liste der Thumbnails eingebunden. Dazu können wir direkt auf das Property `src` des Image-Elements zugreifen. Die Prüfung ist nötig, um das Element nur anzuzeigen, wenn auch wirklich ein Bild zum Buch hinterlegt ist.

Listing 6–18
Das Template der BookListComponent (book-list.component.html)

```
<div class="ui middle aligned selection divided list">
  <a *ngFor="let book of books" class="item">
    <img class="ui tiny image"
     *ngIf="book.thumbnails"
     [src]="book.thumbnails[0].url">
    <div class="content">
      <div class="header">{{ book.title }}</div>
      <div class="description"> {{ book.subtitle }} </div>
      <div class="metadata">
        <span *ngFor="let author of book.authors; last as l">
          {{ author }}<span *ngIf="!l">, </span>
        </span>
      </div>
      <div class="extra">ISBN {{ book.isbn }}</div>
    </div>
  </a>
</div>
```

Beispieldaten

Damit wir auch sofort ein paar Bücher in der Liste sehen können, initialisieren wir in der Komponente ein Array mit Beispielbüchern (Listing 6–19). Wir geben hier alle Parameter an, die der Konstruktor von `Book` erwartet (auch die optionalen), denn im Template der Listenansicht werden ja auch alle diese Werte verwendet. Für das Coverbild des Buchs erzeugen wir ein `Thumbnail`-Objekt, das wir in einem Array übergeben.

Listing 6–19 BookListComponent (book-list.component.ts)

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

import { Book, Thumbnail } from '../shared/book';

@Component({
  selector: 'bm-book-list',
  templateUrl: './book-list.component.html'
})
export class BookListComponent implements OnInit {
  books: Book[];

  ngOnInit() {
    this.books = [
      new Book(
        '9783864903571',
        'Angular',
        ['Johannes Hoppe', 'Danny Koppenhagen',
         ↪ 'Ferdinand Malcher', 'Gregor Woiwode'],
        new Date(2017, 3, 1),
        'Grundlagen, fortgeschrittene Techniken und Best Practices
         ↪  mit TypeScript - ab Angular 4, inklusive NativeScript
         ↪  und Redux',
        5,
        [new Thumbnail('https://ng-buch.de/cover2.jpg',
         ↪ 'Buchcover')],
        'Mit Angular setzen Sie auf ein modernes und modulares...'
      ),
      new Book(
        '9783864901546',
        'AngularJS',
        ['Philipp Tarasiewicz', 'Robin Böhm'],
        new Date(2014, 5, 29),
        'Eine praktische Einführung',
        5,
```

```
        [new Thumbnail('https://ng-buch.de/cover1.jpg',
         ↪ 'Buchcover')],
        'Dieses Buch führt Sie anhand eines zusammenhängenden
         ↪ Beispielprojekts...'      )
    ];
  }
}
```

Der Code für die Initialisierung der Daten soll ausgeführt werden, wenn die Komponente geladen wird. Man könnte dafür den Konstruktor der Klasse verwenden. In Angular kommt allerdings die Methode `ngOnInit()` zum Einsatz, wie im Listing 6–19 zu sehen ist. Diese Methode wird automatisch aufgerufen, wenn die Komponente vollständig initialisiert ist. Sie ist einer der sogenannten *Lifecycle-Hooks* von Angular. Details dazu finden Sie im Kasten auf Seite 85.

ngOnInit() für Initialisierung verwenden

`ngOnInit()` statt Konstruktor – die Lifecycle-Hooks von Angular

Eine Angular-Komponente hat einen definierten Lebenszyklus. Sie wird zunächst initialisiert und es werden ihre Bestandteile gerendert. Wird die Komponente nicht mehr benötigt, wird sie abgebaut und die Ressourcen werden freigegeben.

Mit den Lifecycle-Hooks von Angular können wir gezielt in diesen Lebenszyklus einer Komponente eingreifen. Wechselt die Komponente in einen bestimmten Zustand, können wir Aktionen an dieser Stelle im Ablauf ausführen.

Im Beispiel haben wir die Methode `ngOnInit()` eingesetzt. Sie wird automatisch ausgeführt, wenn die Komponente geladen wurde. Damit die Methode richtig definiert wird, muss eine Klasse immer das Interface `OnInit` implementieren.

Es existieren noch weitere Lifecycle-Hooks, auf die wir an dieser Stelle aber gar nicht näher eingehen wollen. Stattdessen widmen wir uns dem Lebenszyklus von Komponenten ab Seite 500 ausführlicher.

Merke: Statt dem Konstruktor der Klasse sollten wir zur Initialisierung immer die Methode `ngOnInit()` einsetzen.

Komponente ins Template einbinden

Damit ist unsere Listenkomponente komplett. In der Hauptkomponente `AppComponent` muss die neue Komponente nun noch eingebunden werden. Dazu fügen wir ins Template das HTML-Element `<bm-book-list></bm-book-list>` ein. An dieser Stelle finden wir auch das Präfix wieder, das wir beim Anlegen des Projekts angegeben haben.

Ein Präfix verwenden

Die Selektoren von Komponenten und Direktiven sollten in Angular nach Möglichkeit immer mit einem Präfix versehen werden. Das Präfix soll vor allem dafür sorgen, dass Elemente der Anwendung gut von nativen HTML-Elementen unterschieden werden können. Wir beugen damit Konflikten mit anderen Elementen vor, die eventuell dieselbe Bezeichnung verdient hätten.

Beim Anlegen des Projekts mit der Angular CLI haben wir die Option `-p bm` verwendet. Diese Angabe finden wir auch in der Datei `.angular-cli.json` wieder. Wenn wir Komponenten oder Direktiven mit der Angular CLI anlegen, wird das Präfix automatisch im Selektor verwendet.

Das Element wird von Angular durch die Komponente `BookListComponent` ersetzt, denn die Komponente besitzt ja dafür den passenden Selektor `bm-book-list`. Einen Konstruktor haben wir für die Hauptkomponente nicht angelegt, weil die gesamte Initialisierung in `ngOnInit()` vorgenommen wird.

Wir haben uns an dieser Stelle von der Vorgabe der Angular CLI entfernt und das Template *inline* notiert, weil es nur wenige Zeichen benötigt. Sobald das HTML komplexer wird, werden wir das Template wieder in die Datei `app.component.html` auslagern.

Listing 6–20 *Hauptkomponente* AppComponent (app.component.ts)

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

@Component({
  selector: 'bm-root',
  template: '<bm-book-list></bm-book-list>'
})
export class AppComponent { }
```

Bootstrapping

Damit die Anwendung funktioniert, muss die Root-Komponente `AppComponent` von Angular gestartet werden. Die notwendige Anpassung hat die Angular CLI bereits für uns vorgenommen. So wird in der Datei `src/app/app.module.ts` per `@NgModule()` die Root-Komponente »gebootstrappt«. Sie ist dort in der Eigenschaft `bootstrap` in den Modulmetadaten eingetragen.

Wir können die Anwendung nun mit dem Befehl `ng serve` starten. Rufen wir die URL im Browser auf, wird die Liste der Bücher angezeigt.

Angular
Grundlagen, fortgeschrittene Techniken und Best Practices mit TypeScript
Johannes Hoppe, Danny Koppenhagen, Ferdinand Malcher, Gregor Woiwode
ISBN 9783864903571

AngularJS
Eine praktische Einführung
Philipp Tarasiewicz, Robin Böhm
ISBN 9783864901546

Abb. 6–2
Die Buchliste funktioniert.

Was haben wir gelernt?

Abschließend lässt sich feststellen, dass in Angular die Template-Syntax in mehrere Konzepte aufgebrochen wird. Der Datenfluss zwischen Komponenten wird dadurch konkret definiert, und es ist mit einem Blick auf ein Template möglich, zu erkennen, wie sich eine Komponente verhält. Somit können, im Gegensatz zur Vorgängerversion AngularJS, Templates in Angular differenzierter und genauer beschrieben werden.

- Komponenten sind die wichtigsten Bausteine einer Anwendung.
- Eine Komponente besteht immer aus einer TypeScript-Klasse mit Metadaten und einem Template.
- Komponenten werden an Elemente des DOM-Baums gebunden. Das ausgewählte Element wird das Host-Element der Komponente.
- Für die Auswahl der DOM-Elemente wird in der Komponente ein CSS-Selektor festgelegt.
- Angular verfügt über eine eigene Template-Syntax und eingebaute Direktiven, z. B. `ngFor` und `ngIf`.
- Alle Bausteine der Anwendung werden per `@NgModule()` in Angular-Module organisiert. Wir nutzen Imports und Exports, um die Bausteine zu verknüpfen.
- Eine Komponente muss immer in den `declarations` eines Moduls registriert werden.

Demo und Quelltext:
https://ng-buch.de/it1-comp