Kapitel 1
Angular-Kickstart: Ihre erste Angular-Webapplikation

Freuen Sie sich auf Ihre erste Angular-Applikation. Am Beispiel einer Blogging-Plattform werden Sie die wichtigsten Angular-Konzepte kennenlernen und erste Kontakte mit einer Menge neuer und spannender Technologien knüpfen.

Die Angular-Plattform basiert auf einer Vielzahl moderner Technologien und Standards. In diesem Kapitel gebe ich Ihnen zunächst eine Übersicht über diese Technologien und stelle Ihnen die wichtigsten Konzepte des Frameworks vor. Nach Abschluss des Kapitels haben Sie die Kernbestandteile von Angular kennengelernt, Ihre ersten Komponenten in TypeScript verfasst und das Module-Konzept von ECMAScript 2015 verwendet, um aus einzelnen Komponenten eine lauffähige Applikation zu erstellen. Freuen Sie sich also auf eine Menge neuer Themen.

1.1 Installation der benötigten Software

Bevor Sie mit der Implementierung der Applikation beginnen, sollten Sie zunächst einige Werkzeuge installieren. Durch die konsequente Verwendung von *npm* zur Installation von Projektabhängigkeiten halten sich aber die Tools, die Sie benötigen, in sehr überschaubaren Grenzen.

1.1.1 Node.js und npm

Sollten Sie in der Vergangenheit bereits mit JavaScript gearbeitet haben, sind Sie wahrscheinlich schon mit *Node.js* und dem zugehörigen Paketmanager *npm* in Berührung gekommen. Bei *Node.js* handelt es sich um eine Plattform, die ursprünglich entworfen wurde, um serverseitige JavaScript-Applikationen zu ermöglichen. Der *Node Package Manager* (kurz *npm*) ist in diesem Zusammenhang dafür zuständig, die Abhängigkeiten einer Node-Applikation zu verwalten.

Neben diesem Einsatzgebiet hat sich npm mittlerweile aber auch zum Quasistandard für die Installation von Software entwickelt, die in JavaScript implementiert wird. Im

Laufe des Buches werden Sie sowohl Node.js als auch npm näher kennenlernen. Zur Installation der beiden Tools laden Sie sich am einfachsten das für Ihre Plattform passende Installationspaket herunter:

https://nodejs.org

Achten Sie hierbei darauf, dass Sie Node.js mindestens in der Version 14 installieren.

Achtung, Windows-User!

Wenn Sie Node.js unter Windows installieren, kann es bei der Verwendung immer wieder zu unerwartetem Verhalten kommen – insbesondere im Zusammenspiel mit früheren Versionen des .NET-Frameworks. Der Grund hierfür ist meistens nicht Node.js selbst, sondern die Erweiterung *node-gyp*, die für die Kompilierung von nativen Erweiterungen zuständig ist. Sollte dieses Problem auftreten, empfehle ich Ihnen, die Suchbegriffe »node-gyp windows« in die Suchmaschine Ihrer Wahl einzugeben – die ersten Treffer enthalten einige wertvolle Tipps für die Lösung der Probleme. Alternativ kann sich außerdem die Installation einer Linux-Distribution in einer virtuellen Maschine (z. B. VirtualBox) anbieten. Sollten Sie bereits Linux-Erfahrung haben, kann dies der auf Dauer bessere Weg sein.

1.1.2 Visual Studio Code: eine kostenlose Entwicklungsumgebung für TypeScript und Angular

Für die Entwicklung komplexer Webanwendungen ist eine gute Entwicklungsumgebung unerlässlich. Entwicklern steht dafür eine Vielzahl an guten Editoren zur Auswahl, und sollten Sie bereits Ihren persönlichen Favoriten gefunden haben, können Sie ihn natürlich gern weiterhin verwenden.

Allen anderen Lesern möchte ich an dieser Stelle den von Microsoft entwickelten kostenlosen Editor *Visual Studio Code* ans Herz legen. Insbesondere die sehr gute Unterstützung von *TypeScript* (der vom Angular-Team verwendeten Sprache zur Implementierung von Angular-Anwendungen) macht Visual Studio Code zu einem guten Editor für die kommenden Implementierungen. Installationspakete für Ihre Plattform finden Sie unter:

https://code.visualstudio.com

1.1.3 Alternative: Webstorm: perfekte Angular-Unterstützung

Eine (kostenpflichtige) Alternative zu Visual Studio Code ist der Editor *Webstorm*. Neben einer sehr guten TypeScript-Integration bietet er Ihnen zusätzlich noch diverse Annehmlichkeiten speziell für die Angular-Entwicklung. Sollten Sie Angular täglich

bei Ihrer Arbeit einsetzen, kann sich die überschaubare Investition durchaus lohnen. Eine kostenlose 30-Tage-Testversion erhalten Sie über die Internetseite der IDE:

www.jetbrains.com/webstorm

1.2 Hallo Angular

Sie haben nun alle notwendigen Werkzeuge für die Implementierung Ihrer ersten Angular-Anwendung zur Hand.

Bevor es in Abschnitt 1.3, »Die Blogging-Anwendung«, an Ihre erste »echte« Applikation geht, stelle ich Ihnen zunächst ganz klassisch die Grundbausteine einer Angular-Anwendung anhand eines Hello-World-Beispiels vor. Laden Sie sich hierfür, falls noch nicht geschehen, zunächst die Beispielquelltexte von der URL

www.rheinwerk-verlag.de/5285

herunter, und entpacken Sie diese in einen beliebigen Ordner auf Ihrer Festplatte. Öffnen Sie anschließend die Kommandozeile, wechseln Sie in den Unterordner *kickstart/hello-angular*, und führen Sie dort den Befehl

```
npm install
```

aus. Der Node Package Manager (npm) installiert nun alle notwendigen Abhängigkeiten, die in der Datei *package.json* definiert sind. Dieser Vorgang kann insbesondere beim ersten Mal einige Zeit in Anspruch nehmen.

Öffnen Sie das Projekt nun in Ihrer Entwicklungsumgebung. Falls Sie sich für Visual Studio Code entschieden haben, klicken Sie einfach auf File • Open Folder und wählen das entsprechende Verzeichnis aus. Sie sollten nun etwa die in Abbildung 1.1 dargestellte Ansicht sehen.

Zugegebenermaßen wirkt die Struktur für ein »Hello World«-Projekt ziemlich umfangreich – aber keine Angst: Sie werden ein solches Projekt mit großer Wahrscheinlichkeit niemals von Hand aufsetzen müssen. Wie Sie an der im Ordner liegenden Datei *angular.json* erkennen können, basiert die hier vorgestellte Anwendung auf dem Angular-*Command-Line-Interface* (CLI), das ich Ihnen im folgenden Kapitel noch im Detail vorstellen werde.

Hier soll es aber zunächst einmal um die Kernbestandteile einer Angular-Applikation gehen. Im Wesentlichen besteht die hier vorgestellte Anwendung aus den beiden Dateien *app.module.ts* und *app.component.ts* im *app*-Ordner. Zusätzlich dient die Datei *main.ts* als Startpunkt der Applikation (dazu lesen Sie mehr in Abschnitt 1.2.4).

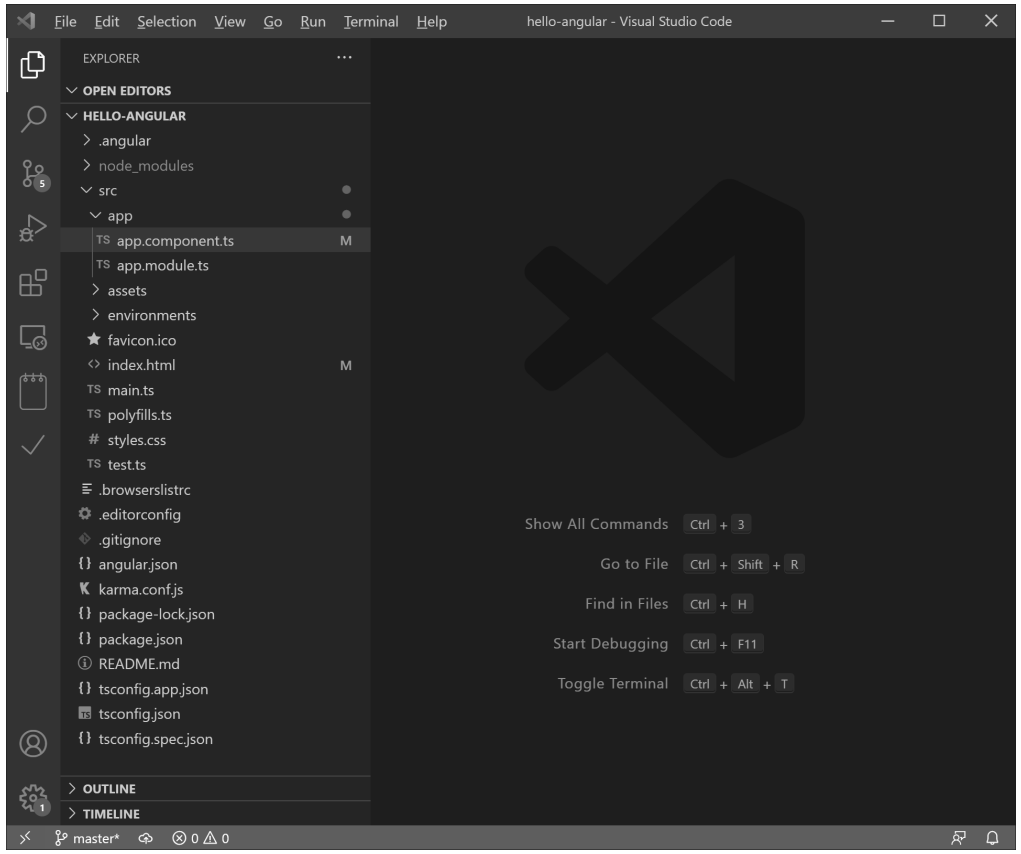

Abbildung 1.1 Verzeichnisstruktur der Hello-World-Anwendung

Öffnen Sie zunächst die eigentliche »Anwendungslogik« in der Datei *app.component.ts*:

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

@Component({
  selector: 'app-root',
  template: `<span>Hallo {{name}}!</span>`
})
export class AppComponent {
  name: string;
  constructor() {
    this.name = 'Angular';
  }
}
```

Listing 1.1 »app.component.ts«: Ihre erste Angular-Komponente

Tada! Sie sehen gerade Ihre erste *Angular-Komponente* vor sich.

Hinweis zur Benennung von Komponenten-Dateien

Bei der Benennung von Komponenten-Dateien hat es sich als gute Praxis etabliert, diese mit der Endung *.component.ts* zu versehen. Auf diese Weise erkennen Sie bereits am Namen, dass die Datei eine Komponente enthält. Wie Sie im folgenden Kapitel sehen werden, unterstützt das Angular-CLI Sie bei der Einhaltung dieser Regeln, indem es die passenden Generatoren bereitstellt.

Sollten Sie in der Vergangenheit mit reinem JavaScript auf Basis von ECMAScript 5 gearbeitet haben, werden Sie in diesem Listing vermutlich direkt eine Reihe von Syntaxelementen entdecken, die Ihnen unbekannt vorkommen. Hierbei handelt es sich um *TypeScript*, eine von Microsoft entwickelte Programmiersprache, die JavaScript unter anderem um Konzepte wie Typsicherheit, Klassen oder Annotationen erweitert. TypeScript ist dabei ein »Superset« von JavaScript.

TypeScript und ECMAScript

Ich werde in den folgenden Abschnitten immer wieder von ECMAScript- und TypeScript-Konzepten sprechen. Dies kann am Anfang etwas verwirrend sein, und ich werde im weiteren Verlauf noch näher auf die genaue Differenzierung der beiden Sprachen bzw. Sprachstandards eingehen.

Für den Anfang reicht es aber, wenn Sie wissen, dass ECMAScript den Sprachstandard von JavaScript beschreibt. Dieser Standard wird jedes Jahr um neue Sprach-Bestandteile ergänzt. So wurde die Unterstützung von Klassen beispielsweise mit ECMAScript 2015 zum Standard hinzugefügt. Die *async/await-Syntax* war Teil von ECMAScript 2020.

TypeScript ist in diesem Zusammenhang eine eigenständige Sprache, die zwar auf dem ECMAScript-Standard (und damit auf JavaScript) aufbaut, diese aber unter anderem um Typ-Sicherheit und Dekoratoren erweitert.

Lassen Sie sich nicht von den einzelnen Begriffen verwirren, sondern freuen Sie sich darauf, die neuen Technologien im Einsatz zu sehen. Im Anhang dieses Buches finden Sie im Übrigen eine detaillierte Vorstellung der wichtigsten Sprachelemente von ECMAScript 2015 bis 2021 und TypeScript.

Im Folgenden wird ECMAScript übrigens manchmal mit »ES« abgekürzt, sodass Sie dann etwas wie »ES2015« lesen.

Wie Sie in Listing 1.1 bereits erkennen können, besteht die Komponente aus zwei Teilen:

- aus dem eigentlichen Komponenten-Code, der durch eine TypeScript-Klasse repräsentiert wird, sowie

- aus dem Decorator (`@Component`), der die Konfiguration und die Anmeldung der Komponente beim Angular-Framework übernimmt.

In Kapitel 3, »Komponenten und Templating: der Angular-Sprachkern«, werden Sie Komponenten und deren weitere Konfigurationsmöglichkeiten noch im Detail kennenlernen.

Doch schauen wir uns die Komponente Schritt für Schritt gemeinsam an. Der erste Ausschnitt, der Ihnen vermutlich neu sein wird, ist die `import`-Anweisung. Mithilfe der Zeile

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

importieren Sie den Decorator `Component` aus dem Modul `'@angular/core'`. Das `import`-Schlüsselwort ist Teil der ES2015-Modulsyntax – einer der zentralen Neuerungen aus dem neuen JavaScript-Standard *ECMAScript 2015*. Im weiteren Verlauf des Buches werden Sie noch tiefergehende Erfahrungen mit der Arbeit mit Modulen sammeln. Zunächst können Sie sich ES2015-Module aber als eine Möglichkeit vorstellen, Ihren Code in gekapselten Codebausteinen zu verwalten und diese bei Bedarf dynamisch zu laden. Durch die obige `import`-Anweisung haben Sie nun also Zugriff auf die importierte Klasse `Component` und können diese in Ihrer Komponente verwenden.

1.2.1 Komponenten konfigurieren

Der nächste Baustein ist der Code-Block:

```
@Component({
  selector: 'app-root',
  template: `
    <span>
      Hallo {{name}}!
   </span>`
})
```

Listing 1.2 Der »Component«-Ausschnitt aus Listing 1.1

Bei diesem Element handelt es sich um einen *TypeScript-Decorator*. Mithilfe von Decorators können Sie Ihre Klassen um zusätzliche Informationen – sogenannte Meta-Daten – erweitern. Angular nutzt diese Meta-Daten anschließend, um die lauffähige Applikation zu erzeugen. Der `@Component`-Decorator ist in diesem Fall die zentrale Stelle zur Konfiguration Ihrer Komponenten.

Konkret teilen Sie Angular über die `selector`-Eigenschaft der Annotation mit, dass Ihre Komponente über das `<app-root>`-Tag in einem HTML-Template instanziiert werden soll.

Das Selektor-Präfix

Bei der Definition der `selector`-Eigenschaft empfiehlt es sich, diese mit einem Präfix zu versehen, um Namenskollisionen mit anderen Bibliotheken zu vermeiden.

Das hier verwendete Präfix `app-` ist natürlich nur bedingt sinnvoll. Vielmehr sollten Sie Ihre persönlichen Initialen (für wiederverwendbare Komponenten) oder eine Abkürzung für Ihr aktuelles Projekt verwenden. Daher werde ich für wiederverwendbare Komponenten ab jetzt das Präfix `ch-` als Abkürzung meines Namens verwenden. Für das durchgehende Projekt-Manager-Beispiel, das ich Ihnen ab Kapitel 8 vorstellen werde, wird das Präfix `pjm-` genutzt. Auf diese Weise können Sie auch im HTML-Code leicht erkennen, welche Komponente Sie an der jeweiligen Stelle verwenden. Das Angular-CLI unterstützt Sie auch an dieser Stelle, indem es alle generierten Komponenten mit dem voreingestellten Selektor-Präfix versieht.

Mithilfe der `template`-Eigenschaft konfigurieren Sie schließlich, dass bei der Instanziierung der Komponente das hinterlegte Template gerendert werden soll.

Außer der Konfiguration des Selektors und des zu rendernden Templates bietet der `@Component`-Decorator noch eine Reihe weiterer Konfigurationsmöglichkeiten, z. B. um die Schnittstelle der Komponente zu definieren. Diese Möglichkeiten werde ich Ihnen im weiteren Verlauf dieses Kapitels sowie im Detail in Kapitel 3, »Komponenten und Templating: der Angular-Sprachkern«, vorstellen.

ECMAScript 2015: Template-Strings

In Listing 1.2 sehen Sie außerdem eine weiteres Sprachelement aus dem ECMAScript-2015-Standards in Aktion: *Template-Strings*. Die Verwendung eines Backticks (`` ` ``) zur Einleitung des Strings ermöglicht Ihnen hier die einfache Definition von mehrzeiligen Strings. Interessant ist dabei außerdem, dass Sie innerhalb solcher Strings Variablen über die Syntax `` `...${var}` `` einfügen können. Anstatt berechnete und mehrzeilige Strings aufwendig mithilfe des +-Operators aneinanderzufügen, können Sie diese somit sehr elegant mithilfe von Template-Strings definieren.

Das eigentliche Template ist in diesem Beispiel zugegebenerweise trivial. Dennoch sehen Sie hier bereits ein spezielles Element der Templating-Syntax in Aktion, nämlich die *Interpolation*:

```
Hallo {{name}}!
```

Mithilfe der geschweiften Klammern teilen Sie Angular mit, dass Sie an dieser Stelle den Wert der Instanzvariablen `name` aus der zugehörigen Komponenten-Klasse rendern möchten. AngularJS-1.x-Usern sollte diese Syntax bereits sehr vertraut vorkommen. In der neuen Angular-Plattform bildet diese Art der Interpolation aber lediglich eine syntaktische Vereinfachung für das noch deutlich mächtigere *Property-Binding*, das Sie ebenfalls im weiteren Verlauf des Kapitels noch näher kennenlernen werden.

1.2.2 Die Komponenten-Klasse

Die Implementierung der Komponenten-Logik erfolgt nun wie bereits angekündigt in Form einer *Klasse*. Über den Codeblock

```
export class AppComponent {
  ...
}
```

definieren Sie die Klasse `AppComponent`. Das vorangestellte Schlüsselwort `export` ist ebenfalls Teil der ECMAScript-2015-Modulsyntax und sorgt dafür, dass die Klasse von anderen Modulen importiert und verwendet werden kann.

Neben der Modulsyntax ist die dedizierte Unterstützung von Klassen ein weiterer zentraler Sprachbestandteil aus dem ECMAScript-2015-Standard. Wurden objektorientierte Sprachelemente wie Klassendefinitionen, Vererbung oder Konstruktoren in der Vorgängerversion von ECMAScript2015 (ECMAScript 5) noch recht aufwendig über die Verwendung von Prototypen simuliert, bietet ES2015 eine deutlich intuitivere Syntax für die Definition von Klassen.

So zeigt Listing 1.3 die vollständige Klassendefinition mit der Membervariablen `name`. Das Schlüsselwort `constructor` definiert den Konstruktor, der bei der Instanziierung eines `AppComponent`-Objekts aufgerufen wird und die Membervariable `name` initialisiert:

```
export class AppComponent {
  name: string;
  constructor() {
    this.name = 'Angular';
  }
}
```

Listing 1.3 Einsatz der neuen Klassensyntax

Im Vergleich dazu hätte die äquivalente Implementierung der Klasse `AppComponent` in ES5 wie folgt ausgesehen:

```
AppComponent = (function () {
  function AppComponent() {
    this.name = 'Angular';
  }
  return AppComponent;
})();
```

Listing 1.4 Klassendefinition unter ECMAScript 5

In Bezug auf Lesbarkeit und Verständlichkeit machte der ES2015-Standard einen deutlichen Schritt nach vorne.

TypeScript – Typsicherheit für JavaScript-Anwendungen

Eine Besonderheit, die ich bislang übergangen habe, besteht außerdem in der Deklaration der `name`-Membervariablen. So handelt es sich bei dem Statement

```
name: string;
```

um eine TypeScript Typen-Deklaration. Während der ECMAScript-Standard bis heute weiterhin typenlos ist, erweitert TypeScript den Sprachumfang um die Definition von statischen Typen. Die obige Deklaration teilt dem TypeScript-Compiler mit, dass es sich bei der `name`-Variablen um einen String handelt. Die Anweisung

```
name = 4;
```

würde schon beim Kompiliervorgang zu einem Fehler führen:

```
hello.ts(15,9): error TS2322: Type 'number' is not assignable to type
'string'.
```

Die große Besonderheit im Gegensatz zu anderen statisch typisierten Sprachen wie Java oder C# ist dabei, dass Sie nicht dazu gezwungen werden, Typen einzusetzen. Da TypeScript neben der Typisierung noch eine Vielzahl an weiteren sehr interessante Sprachfeatures bereithält, habe ich mich entschieden, Ihnen die Sprache in Anhang B, »Typsicheres JavaScript mit TypeScript«, im Detail vorzustellen.

1.2.3 Das Applikationsmodul: das Hauptmodul der Anwendung konfigurieren

Sie haben nun also eine vollständig implementierte Angular-Komponente. Damit ist es an der Zeit, sich dem zweiten Kernbestandteil einer jeden Angular-Anwendung zu widmen: dem Applikationsmodul. Öffnen Sie hierfür die Datei *app.module.ts*:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';

@NgModule({
  imports: [BrowserModule],
  declarations: [AppComponent],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Listing 1.5 »app.module.ts«: Implementierung des Applikationsmoduls

Ähnlich wie der `@Component`-Decorator bietet Ihnen der `@NgModule`-Decorator die Möglichkeit, eine deklarative Konfiguration eines Angular-Moduls vorzunehmen. Mithilfe von Modulen können Sie in der Angular-Plattform zusammengehörige Komponenten und Services kapseln. Dabei besteht jede Angular-Applikation aus mindestens einem Modul – dem Applikationsmodul.

Zur Definition eines Applikationsmoduls, das im Browser laufen soll, müssen Sie dem Decorator nun zwei Dinge mitteilen: die Einstiegskomponente in die Applikation (`bootstrap: [ AppComponent ]`) sowie die Tatsache, dass Sie die Funktionalität aus dem `BrowserModule` verwenden möchten (`imports: [ BrowserModule ]`).

Angular: Nicht nur für Browser-Anwendungen

Auch wenn sich dieses Buch auf die Implementierung von Anwendungen für den Browser konzentrieren wird, ist die grundlegende Architektur der Angular-Plattform darauf ausgelegt, ebenso gut andere Plattformen zu unterstützen. So stellt Ihnen das Framework *NativeScript* (*https://nativescript.org/*) beispielsweise die Möglichkeit zur Verfügung, native Mobile-Anwendungen für Android und iOS auf Basis von Angular zu entwickeln.

Der Import des `BrowserModule` im Beispielprojekt sorgt an dieser Stelle dafür, dass als Ergebnis Ihres Angular-Codes ein *DOM-Baum* gerendert wird, wohingegen *NativeScript* eine native Android-View oder iOS-View erzeugen würde.

Die letzte noch nicht behandelte Eigenschaft ist die `declarations`-Eigenschaft. Alle hier aufgeführten Komponenten sind *im gesamten Applikationsmodul* sichtbar und können von jeder anderen Komponente verwendet werden, die hier deklariert wird. Da die HelloWorld-Applikation lediglich aus einer einzigen Komponente besteht, wirkt die Konfiguration zugegebenermaßen etwas »aufgeblasen«. Insbesondere bei der Entwicklung von umfangreicheren Anwendungen werden Sie diesen Ansatz aber schnell zu schätzen lernen!

NgModule vs. ECMAScript-Modulsyntax

Die Namenswahl `NgModule` als Begriff für die Definition von in sich gekapselten Applikationseinheiten wurde in der Angular-Community lange Zeit kontrovers diskutiert. So bestand einer der Hauptkritikpunkte an der Benennung darin, dass der Begriff leicht mit der ECMAScript-Modulsyntax verwechselt werden könne, die ebenfalls für viele Entwickler neu war.

Es ist also wichtig zu verstehen, dass es sich bei Angular-Modulen (`@NgModule`) und ECMAScript-Modulen (`import`/`export`-Syntax) um zwei voneinander völlig unabhängige Konzepte handelt. Während ECMAScript-Module (Angular-unabhängig) für die Strukturierung Ihrer Quelltexte zuständig sind, bieten `NgModule`-Module Ihnen die

Möglichkeit, Ihre Angular-Anwendung in logisch zusammenhängende Einheiten zu unterteilen.

In Kapitel 15, »NgModule und Lazy-Loading: Modularisierung Ihrer Anwendungen«, werden Sie des Weiteren sehen, wie Sie mithilfe von Angular-Modulen bestimmte Teile Ihrer Anwendung erst bei deren Verwendung nachladen können (Stichwort *Lazy-Loading*).

1.2.4 main.ts: Wahl der Ausführungsplattform und Start des Applikationsmoduls

Der echte Start der Applikation erfolgt schließlich in der Datei *main.ts* durch die Übergabe des `AppModule` an die Plattform, die Sie verwenden wollen:

```
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {AppModule} from './app/app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```

Listing 1.6 »main.ts«: Start des Hauptmoduls über die Methode »bootstrapModule«

main.ts und app.module.ts

Auch wenn Sie in der Benennung Ihrer Dateien grundsätzlich frei sind, hat es sich bewährt, die Dateien für den Start der Anwendung und für die Definition des Haupt-Applikationsmoduls *main.ts* und *app.module.ts* zu nennen. Ich möchte Ihnen diese Konvention ans Herz legen: So werden Sie und andere Entwickler sich wesentlich leichter in Ihren Anwendungen zurechtfinden.

1.2.5 index.html: Einbinden der Bootstrap-Komponente und Start der Anwendung

Sie haben es fast geschafft. Sie müssen nur noch die Anwendung in die *index.html* einbinden und von dort aus den Bootstrap-Vorgang auslösen. Listing 1.7 zeigt die entsprechende Implementierung aus dem Hello-Angular-Projekt:

```
<html>
    <head>
        <title>Hello Angular</title>
        <base href="/">
    </head>
    <body>
```

```
        <!-- Einbinden der Hello-Komponente -->
        <app-root>Die Applikation wird geladen...</app-root>
    </body>
</html>
```

Listing 1.7 Aufbau der »index.html« als Einstiegspunkt für Ihre Applikation

Der erste Code-Baustein, der Ihnen vermutlich ins Auge gesprungen ist, ist die Verwendung des `app-root`-Tags:

```
<!-- Einbinden der Hello-Komponente -->
<app-root>Die Applikation wird geladen ...</app-root>
```

Listing 1.8 »index.html«: Einstieg in die Applikation über das »app-root«-Tag

Wie bereits beschrieben, sorgt die `selector`-Konfiguration der `AppComponent`-Komponente dafür, dass an dieser Stelle Ihre Komponente gerendert wird. Der Body des Tags wird nach erfolgreichem Laden der Komponente ersetzt, sodass Sie innerhalb des Tags die Möglichkeit haben, einen Ladeindikator oder Ähnliches einzubinden.

Bei genauerem Hinsehen wird Ihnen aber auch in dieser Datei eine interessante Änderung im Vergleich zur traditionellen JavaScript-Entwicklung auffallen: Die Datei *index.html* enthält kein einziges `<script>`-Tag zum Einbinden Ihrer Anwendung.

Der Grund hierfür ist, dass das Angular-CLI die hier einzubindenden Skripte erst während des Builds der Anwendung erzeugt und diese dann dynamisch in die *index.html* einfügt, die an den Browser ausgeliefert wird. Dies hat den großen Vorteil, dass Sie sich selbst keinerlei Gedanken um eine sinnvolle Paketierung und Auslieferung der Applikation machen müssen: Weitere Details hierzu werde ich Ihnen in Kapitel 2, »Das Angular-CLI: professionelle Projektorganisation für Angular-Projekte«, vermitteln.

Um Ihre Anwendung auszuführen, wechseln Sie nun auf der Kommandozeile in das entsprechende Wurzelverzeichnis der Applikation und führen dort den Befehl

```
npm start
```

aus. Im Hintergrund passieren nun zwei Dinge:

1. Das Angular-CLI kompiliert Ihre Anwendung mithilfe des TypeScript-Compilers in für den Browser verständlichen JavaScript-Code.
2. Es wird ein einfacher Webserver gestartet, der Ihre fertig kompilierte Anwendung an den Browser ausliefert, das Sourcecode-Verzeichnis in der Folge auf Änderungen hin überwacht und in diesem Fall ein Neuladen des Browserfensters auslöst.

Sie sollten nun eine Anzeige wie in Abbildung 1.2 sehen.

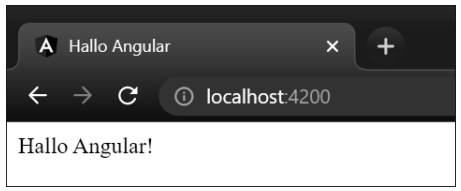

Abbildung 1.2 Ihre erste lauffähige Applikation

Zugegebenermaßen war dies vermutlich eines der längsten Hello-World-Beispiele in der Geschichte der Webentwicklung. Doch mit dem Wissen, das Sie bis hierher erworben haben, besitzen Sie bereits alles Rüstzeug für die Implementierung von komplexeren Angular-Anwendungen. Freuen Sie sich also nun darauf, Ihre erste »echte« Applikation zu implementieren.

1.3 Die Blogging-Anwendung

Abbildung 1.3 zeigt einen Screenshot der Blogging-Anwendung, die Sie in diesem Kapitel entwickeln werden.

Die Anwendung bietet Ihnen die Möglichkeit, neue Blog-Artikel zu verfassen. Die verfassten Artikel werden anschließend in einer Listenansicht dargestellt.

Abbildung 1.3 Die Blogging-Anwendung

Während der Entwicklung dieser Anwendung werden Sie einige neue Techniken einsetzen. Unter anderem werden Sie lernen, wie Sie

- Informationen aus Ihrer View in Ihr Applikationsmodell übertragen (Event-Binding).
- Daten aus Ihrem Applikationsmodell in der View anzeigen (Property-Binding).
- lokale Variablen innerhalb Ihrer View definieren.
- über Listen von Objekten iterieren.
- einzelne Bestandteile Ihrer Anwendung in eigenen Komponenten kapseln.

Sollten Sie die einzelnen Implementierungsschritte im Folgenden selbstständig durchführen wollen, öffnen Sie nun den Ordner *kickstart/blog-start* in Ihrer Entwicklungsumgebung. Dieses Projekt ist insofern vorbereitet, als dass Sie sich nicht mehr um den Import der richtigen Bibliotheken und das Laden der Module kümmern müssen. Ebenso habe ich bereits ein halbwegs ansehnliches CSS-Stylesheet bereitgestellt – Sie können sich somit voll und ganz auf die eigentliche Implementierung der Angular-Komponenten konzentrieren.

Sollten Sie es eher vorziehen, die einzelnen Schritte an der fertigen Lösung nachzuvollziehen, verwenden Sie den Ordner *blog-complete*.

Installation der Abhängigkeiten

Vergessen Sie an dieser Stelle nicht, die benötigten Abhängigkeiten zu installieren. Wie im vorigen Beispiel erledigen Sie dies mithilfe des Befehls `npm install` (oder kurz `npm i`) im jeweiligen Projektverzeichnis.

Wenn Sie sich dafür entschieden haben, das Beispiel Schritt für Schritt selbst zu implementieren, sollten Sie nun etwa die Ansicht aus Abbildung 1.4 sehen.

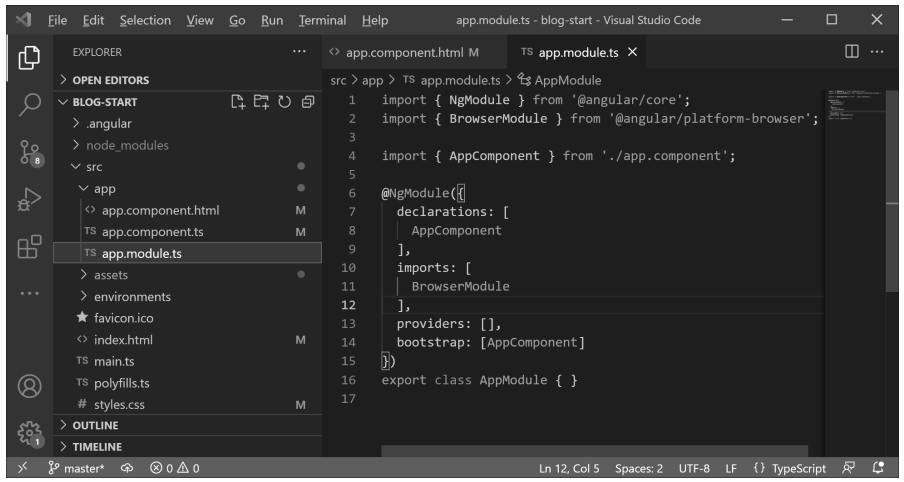

Abbildung 1.4 Die Struktur der Blogging-Anwendung

Wie schon im vorigen Beispiel liegen die Quelltexte der Anwendung im Ordner *app*.

Öffnen Sie als Nächstes die Datei *app.component.ts*, die die Hauptkomponente Ihrer Applikation repräsentiert.

Hinweis zur Dateibenennung

Ich werde im weiteren Verlauf des Buches aus Gründen der Übersichtlichkeit darauf verzichten, jedes Mal den vollständigen Pfad zu einer Quelltextdatei zu nennen. Alle besseren Entwicklungsumgebungen bieten die Möglichkeit, Dateien mit einer bestimmten Tastenkombination über ihren Namen zu öffnen.

Möchten Sie in Visual Studio Code beispielsweise die Datei *app.component.ts* öffnen, so müssen Sie hierfür lediglich die Tastenkombination Strg + P drücken. Über das nun sichtbare Suchfenster (siehe Abbildung 1.5) können Sie sehr komfortabel zu beliebigen Dateien navigieren.

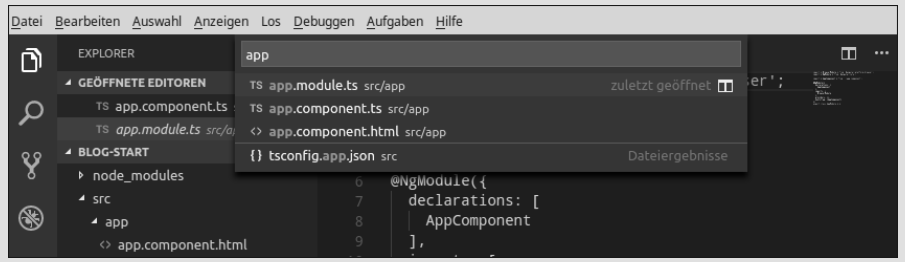

Abbildung 1.5 Suchfenster zum Öffnen von Dateien über ihren Namen

Hier wird Ihnen direkt eine Neuerung im Vergleich zum Hello-World-Beispiel auffallen. Anstatt das Template direkt in der Komponentenkonfiguration zu definieren, wird in diesem Fall über die `templateUrl`-Eigenschaft auf die Datei *app.component.html* verwiesen, die das Template beinhaltet:

```
import {Component} from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  createBlogEntry(title: string, image: string, text: string) {
    console.log(title, image, text);
  }
}
```

Listing 1.9 »app.component.ts«: Verwendung der »templateUrl«-Eigenschaft

Ob Sie Ihre Templates direkt in der Annotation definieren oder diese in eigene HTML-Dateien auslagern, ist eine Frage des Geschmacks. Bei umfangreicheren Templates kann die Definition innerhalb der Annotation aber schnell unübersichtlich werden. Andererseits ist es für das Verständnis der Komponente oft hilfreich, sowohl das Markup als auch die Komponentenlogik an einer Stelle zu haben.

Meine persönliche Empfehlung ist es, bei kleineren gekapselten Hilfskomponenten auf die Möglichkeit der direkten Definition innerhalb der Komponente zurückzugreifen. Bei der Entwicklung von umfangreicheren Komponenten bietet sich hingegen die Auslagerung in eine eigene HTML-Datei an.

ECMAScript-Methoden

Neben der Definition der Template-URL enthält die Komponenten-Klasse noch ein weiteres Sprachelement, das im Zuge von ECMAScript 2015 zum Sprachstandard hinzugefügt wurde. So wird über den Codeblock

```
createBlogEntry(title: string, image: string, text: string) {
  console.log(title, image, text);
}
```

die *Methode* `createBlogEntry` definiert. Klassenmethoden ermöglichen es Ihnen, öffentliche Methoden einer Klasse – und somit deren Schnittstelle – zu definieren. Für die ES5-Entwickler unter Ihnen zeigt der folgende Ausschnitt die äquivalente Implementierung im alten Standard:

```
AppComponent = (function () {
  function AppComponent() {
  }
  AppComponent.prototype.createBlogEntry=function (title, image, text) {
    console.log(title, image, text);
  };
  return AppComponent;
})();
```

1.3.1 Start der Applikation

Um die vorbereitete Anwendung auszuführen, wechseln Sie wie gewohnt auf der Kommandozeile in den jeweiligen Ordner und führen dort den Befehl

```
npm start
```

aus. Wie schon in Abschnitt 1.2 werden die Quelltexte in der Folge auf Änderungen überwacht. Unter der Adresse *http://localhost:4200* sollte nun der vorbereitete Rumpf der Applikation starten (siehe Abbildung 1.6).

Abbildung 1.6 Ausgangssituation für die weitere Entwicklung des Blogs

1.3.2 Einige Tipps zur Fehlersuche

Sollte sich dieses Bild bei Ihnen nicht zeigen, kann dies unterschiedliche Ursachen haben. Der beste Weg, um dem Fehler auf die Schliche zu kommen, führt über die Entwicklerkonsole Ihres Browsers.

Chrome bietet Ihnen in diesem Zusammenhang mit den *Developer-Tools* eine Sammlung von Tools, die es Ihnen als Entwickler beispielsweise ermöglichen, den Netzwerkverkehr zu überwachen oder die Konsolenausgaben Ihrer Applikation einzusehen. Um die Developer-Tools zu öffnen, klicken Sie mit der rechten Maustaste in das Browserfenster und wählen den Menüeintrag UNTERSUCHEN (oder in älteren Versionen des Browsers ELEMENT UNTERSUCHEN). Insbesondere der Reiter CONSOLE kann Ihnen bei der Fehlersuche wertvolle Informationen liefern.

```
▶ERROR Error: The selector "app-roote" did not match any elements
    at DefaultDomRenderer2.selectRootElement (platform-browser.js:686)
    at locateHostElement (core.js:9519)
    at ComponentFactory$1.create (core.js:24805)
    at ApplicationRef.bootstrap (core.js:29235)
    at core.js:28948
    at Array.forEach (<anonymous>)
    at PlatformRef._moduleDoBootstrap (core.js:28948)
    at core.js:28918
    at ZoneDelegate.invoke (zone-evergreen.js:364)
    at Object.onInvoke (core.js:28301)
```

Abbildung 1.7 Anzeige in der Developer-Konsole bei fehlerhaftem Selektor der Root-Komponente

Abbildung 1.7 zeigt Ihnen exemplarisch die Einträge in der Developer-Konsole, die erscheinen, wenn Sie in der Bootstrap-Komponente einen Selektor definiert haben, der nicht in der *index.html* enthalten ist (in diesem Fall `app-roote` anstelle des in der *index.html* verwendeten `<app-root>`). Überprüfen Sie in diesem Fall, ob es Probleme bei der TypeScript-Kompilierung gegeben hat oder ob Sie eventuell einen Tippfehler in Ihrem Quelltext haben.

Außer zur Fehlersuche werden Sie die Konsolenausgabe in den kommenden Kapiteln zur Entwicklung und zum Debugging Ihrer Applikationen verwenden.

1.3.3 Die Formular-Komponente: Daten aus der View in die Komponenten-Klasse übertragen

Im ersten Schritt werden Sie nun lernen, wie Sie mithilfe von *lokalen Variablen* und *Event-Bindings* Daten aus der Oberfläche in Ihre Komponenten-Klasse übertragen können. Sie werden hierfür das Formular aus Abbildung 1.3 implementieren. Im Anschluss daran werden Sie dann lernen, wie Sie Daten mithilfe von *Property-Bindings* aus der Komponenten-Klasse auslesen und in der Oberfläche darstellen können.

Listing 1.10 zeigt die HTML-Implementierung der Formular-Komponente:

```
<div class="form">
  <div class="control">
    <label for="title">Titel:</label>
    <input type="text" id="title" #title/>
  </div>
  <div class="control">
    <label for="image">Bild-URL:</label>
    <input type="text" id="image" #image/>
  </div>
  <div class="control">
    <label for="text">Text:</label>
    <textarea id="text" cols="20" rows="3" #text></textarea>
  </div>
  <div>
 <button (click)="createBlogEntry(title.value, image.value, text.value)">
      Blog-Eintrag anlegen
    </button>
  </div>
</div>
```

Listing 1.10 »app.component.html«: HTML-Implementierung des Formulars

In diesem Listing erkennen Sie erneut zwei neue Syntaxelemente: die Deklaration der drei lokalen Variablen `title`, `image` und `text` sowie die Definition eines Event-Bindings am Button.

Lokale Template-Variablen

Angular ermöglicht es Ihnen, für jedes Element im DOM-Baum eine lokale Variable anzulegen. Die Deklaration einer Variablen erfolgt dabei über ein vorangestelltes #-Zeichen. Diese Variable steht Ihnen dann innerhalb des gesamten Templates zur Verfügung. Über den Ausdruck

```
<input type="text" id="title" #title/>
```


definieren Sie somit die lokale Variable `title`, die sich auf das DOM-Element `<input>` bezieht. In der Folge können Sie nun die gesamte öffentliche DOM-API des Input-Felds verwenden. Hierbei ist es wichtig zu verstehen, dass der Zugriff auf die Eigenschaften des Elements über DOM-Propertys und Methoden (und nicht über DOM-Attribute) geschieht. Eine detaillierte Beschreibung der Unterschiede der Konzepte finden Sie in Kapitel 3, »Komponenten und Templating: der Angular-Sprachkern«.

Event-Bindings

Event-Bindings erlauben es – wie der Name schon vermuten lässt –, auf Events von Komponenten zu reagieren. Die Syntax für die Erstellung eines Event-Bindings hat dabei folgenden Aufbau:

```
(eventName)="auszuführendes Statement"
```

Im Fall von Standard-DOM-Elementen sind die verfügbaren Events ebenfalls über die DOM-API definiert. Die Zeile

```
<button (click)="createBlogEntry(title.value, image.value, text.value)">
```

sorgt somit dafür, dass bei einem Klick auf den Button die im TypeScript-Code definierte Methode `createBlogEntry` aufgerufen wird. In der Parameterliste werden die Werte der drei Input-Felder durch das Auslesen der `value`-Property an die Methode übergeben. Konzeptionell können Sie sich Event-Bindings also als Möglichkeit vorstellen, Daten aus Ihrer View in die Komponenten-Klasse zu übertragen. Sie sollten das Formular jetzt in Ihrem Browser sehen. Nach dem Ausfüllen der drei Felder und einem Klick auf den Button wird (wie in der Methode `createBlogEntry` definiert) der erzeugte Eintrag in der Developer-Konsole ausgegeben (siehe Abbildung 1.8).

Abbildung 1.8 Ausgabe des Blog-Eintrags in der Developer-Konsole

Sie haben nun die Übertragung der Daten von der View in die Komponenten-Klasse implementiert. Der nächste Schritt besteht jetzt darin, die Daten zu speichern und anschließend in der View als Liste darzustellen.

1.3.4 Das Applikationsmodell

Sie werden im Laufe des Buches noch diverse Techniken zum Speichern Ihrer Applikationsdaten und zu deren Verwaltung kennenlernen. So werden Sie beispielsweise in Kapitel 11, »HTTP: Anbindung von Angular-Applikationen an einen Webserver«, lernen, wie Sie HTTP-Backends an Ihre Anwendung anbinden können. Das Beispiel aus diesem Abschnitt beschränkt sich aber zunächst auf die Speicherung der Daten innerhalb der `AppComponent`.

Legen Sie dafür zunächst eine neue Datei mit dem Namen *blog-entry.ts* im Unterordner *models* an, und definieren Sie dort die folgende TypeScript-Klasse zur Kapselung der Eigenschaften eines Blog-Eintrags:

```
export class BlogEntry {
  title = '';
  text = '';
  image = '';
}
```

Listing 1.11 »blog-entry.ts«: Definition der Modell-Klasse »BlogEntry«

Kurz-Exkurs: TypeScript Type-Inference

In Listing 1.11 sehen Sie ein weiteres sehr interessantes TypeScript-Sprachfeature in Aktion: die *Type-Inference*. So hätten Sie die Definition der `title`-Member-Variablen auch wie folgt vornehmen können:

```
export class BlogEntry {
  title: string = '';
  ...
}
```

Die explizite Angabe des Typs `string` ist hier jedoch überflüssig, da TypeScript durch die Zuweisung des Leer-Strings als Default-Wert den Typ für `title` automatisch als `string` festlegt!

Über das `export`-Schlüsselwort wird hier festgelegt, dass die Klasse in andere Dateien importiert werden kann. Die Verwendung in der `AppComponent` sieht somit wie folgt aus:

```
import {BlogEntry} from './models/blog-entry';
...
```



```
export class AppComponent {
  entries: Array<BlogEntry> = [];
  createBlogEntry(title: string, image: string, text: string) {
    const entry = new BlogEntry();
    entry.title = title;
    entry.image = image;
    entry.text = text;
    this.entries.push(entry)
  }
}
```

Listing 1.12 »app.component.ts«: Verwendung der »BlogEntry«-Klasse in der »AppComponent«

Mit der Zeile

```
entries: Array<BlogEntry> = [];
```

definieren Sie zunächst die Membervariable `entries`. Bei der Klasse `Array` handelt es sich um eine TypeScript-Klasse zur Speicherung von Listen. Mithilfe der spitzen Klammern < > teilen Sie TypeScript mit, dass Sie lediglich `BlogEntry`-Instanzen in dieser Liste speichern werden. Die eckigen Klammern am Ende `[]` initialisieren die Liste schließlich als leere Liste.

»const«- und »let«-Variablen

Beim Schlüsselwort `const` handelt es sich erneut um eine Erweiterung aus dem ES2015-Standard. `const` ermöglicht es Ihnen, unveränderliche Block-Scope-Variablen zu definieren, also solche Variablen, die nur im aktuellen Block gültig sind und denen Sie nur einmalig einen Wert zuweisen können. Möchten Sie eine veränderliche Block-Scope-Variable definieren, erfolgt dies über das Schlüsselwort `let`.

Ich stelle Ihnen `const`- und `let`-Variablen im Detail in Anhang A, »ECMAScript 2015 (and beyond)«, vor. Grundsätzlich sollten Sie `const`- und `let`-Variablen gegenüber den in ES5 verwendeteten `var`-Variablen bevorzugen!

Neben der direkten Angabe der Array-Klasse unterstützt TypeScript alternativ noch die Möglichkeit, eine Liste von Werten durch `[]`-Klammern zu deklarieren. Das folgende Statement ist somit äquivalent zur vorherigen Definition:

```
entries: BlogEntry[] = [];
```

Die Befüllung der `entries`-Variablen erfolgt erwartungsgemäß in der `createBlogEntry`-Methode. So wird zunächst eine neue Instanz der Klasse `BlogEntry` erzeugt. Anschließend werden die Eigenschaften der Klasse gefüllt, und das Objekt wird der `entries`-Liste hinzugefügt:

```
createBlogEntry(title: string, image: string, text: string) {
  const entry = new BlogEntry();
  entry.title = title;
  entry.image = image;
  entry.text = text;
  this.entries.push(entry);
}
```

Typsicherheit – Brauche ich das wirklich?

Wenn Sie bislang mit reinem JavaScript gearbeitet haben, wird Ihnen die Arbeit mit Typen vielleicht zunächst etwas aufwendig vorkommen – insbesondere in sehr kleinen Projekten scheint der Nutzen den Mehraufwand für die Definition auf den ersten Blick oft nicht zu rechtfertigen. Aber in solchen Projekten, in denen mehrere Personen am Quelltext arbeiten oder in denen eine Vielzahl von unterschiedlichen Klassen zum Einsatz kommt, kann die statische Typisierung Ihnen durchaus bei der Entwicklung helfen.

Stellen Sie sich beispielsweise vor, Sie (oder andere Entwickler) möchten die Variable `title` in `heading` umbenennen. Ohne die Definition eines Typs müssten Sie nun alle Dateien Ihres Projekts nach dem String `title` durchsuchen und diesen ersetzen. Sollten Sie die Eigenschaft `title` aber ebenfalls in einem anderen Kontext – etwa bei der Definition eines Kalendereintrags – verwendet haben, dürfen Sie dieses Vorkommen selbstverständlich nicht ersetzen: Willkommen in der Refactoring-Hölle!

Die Arbeit mit statischen Typen erleichtert Ihnen ein solches Refactoring ungemein: Denn nach der Umbenennung der Variablen wird Ihnen der TypeScript-Compiler sofort sämtliche Stellen melden, an denen Sie noch mit der `title`-Eigenschaft arbeiten:

```
app.component.ts(25,15): error TS2339:
   Property 'title' does not exist on type 'BlogEntry'.
```

Des Weiteren wird Ihnen Ihre Entwicklungsumgebung die entsprechenden Stellen als fehlerhaft markieren und Ihnen über das Autocomplete-Feature sogar Vorschläge für die Korrektur unterbreiten. Abbildung 1.9 zeigt die Oberfläche des Visual-Studio-Code-Editors während des Refactorings.

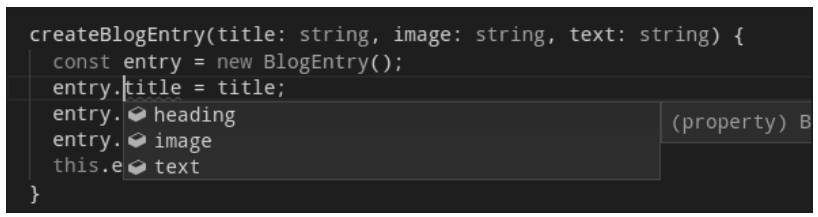

Abbildung 1.9 Fehlermeldung und Autocomplete während des Refactorings

Sie sehen also: Ein bisschen Mehraufwand bei der initialen Deklaration kann Ihnen in der Folge eine Menge Arbeit und Ärger ersparen.

Alternative: TypeScript-Interfaces

In Kapitel 8, »Template-driven Forms: einfache Formulare auf Basis von HTML«, und in Anhang B, »Typsicheres JavaScript mit TypeScript«, werden Sie mit Interfaces eine alternative Technik zur Sicherstellung von Typsicherheit kennenlernen. Dort erfahren Sie auch, welche Vor- und Nachteile der Einsatz der jeweiligen Technik (Klasse oder Interface) hat.

1.3.5 Darstellung der Liste in der View

Der letzte Schritt besteht darin, die in Ihrer Klasse definierten Daten wieder in der View darzustellen. Hierfür benötigen Sie zwei neue Sprachelemente:

1. die `NgFor`-*Direktive* zum Iterieren über eine Liste von Elementen
2. das *Property-Binding*, um Daten aus dem Applikationsmodell an Ihre View-Elemente zu binden

Listing 1.13 zeigt die beiden Techniken im Einsatz. Platzieren Sie diesen Code einfach oberhalb der Formulardefinition in der Datei *app.component.html*:

```
<div class="blog-entry" *ngFor="let entry of entries">
  <div class="blog-image">
    <img [src]="entry.image"/>
  </div>
  <div class="blog-summary">
    <span class="title">{{entry.title}}</span>
    <p>{{entry.text}}</p>
  </div>
</div>
```

Listing 1.13 »app.component.html«: HTML-Implementierung der Listendarstellung

Die Definition der Schleife erfolgt wie bereits erwähnt mithilfe von `NgFor`. Über den Ausdruck

```
*ngFor="let entry of entries"
```

iterieren Sie über die TypeScript-Variable `entries`. Die jeweiligen Eigenschaften der Einträge stehen Ihnen innerhalb der Schleife unter der Variablen `entry` zur Verfügung. Das Schlüsselwort `let` ist in dieser Syntax erneut an den ES2015-Standard angelehnt, der es ermöglicht, über `let` *Block-Scope*-Variablen zu definieren (siehe Anhang A, »ECMAScript 2015 (and beyond)«).

Hinweis zu Verweisen auf Angular-Direktiven

Im weiteren Verlauf des Buches werde ich bei Verweisen auf von Angular mitgelieferte Direktiven immer den Klassennamen der Direktive im Angular-Quelltext (also z. B. `NgFor`) und nicht den von der Direktive verwendeten Selektor (`ngFor`) verwenden. Als Faustregel gilt hier: Der Selektor einer Direktive entspricht dem Klassennamen in *lowerCamelCase*-Schreibweise (also mit einem kleinen ersten Buchstaben). Details zu Selektoren finden Sie in Abschnitt 3.2.

Eine Besonderheit bei dieser Deklaration ist außerdem die Verwendung der *-Syntax. Bei dieser Schreibweise handelt es sich um eine syntaktische Vereinfachung von Template-Elementen. Ich werde Ihnen dieses Konzept in Kapitel 3, »Komponenten und Templating: der Angular-Sprachkern«, noch im Detail erläutern. Vereinfacht ausgedrückt: Sie werden die *-Syntax immer dann einsetzen, wenn Sie durch ihre Verwendung neue DOM-Elemente in den DOM-Baum einfügen möchten. Außer bei der `*ngFor`-Schleife kommt diese Syntax beispielsweise auch bei der `*ngIf`-*Anweisung* zum Einsatz, die abhängig von einer Bedingung DOM-Elemente erzeugt.

Innerhalb der Schleife erfolgt die Ausgabe von `title` und `text` wie im Hello-World-Beispiel durch Interpolation der entsprechenden Werte:

```
<div class="blog-summary">
  <span class="title">{{entry.title}}</span>
  <p>{{entry.text}}</p>
</div>
```

Für die Darstellung des Bildes kommt hingegen das schon angesprochene *Property-Binding* zum Einsatz.

Die Zeile

```
<img [src]="entry.image"/>
```

bindet dabei die DOM-Property `src` des `HTMLImageElement` an die Eigenschaft `image` aus dem `BlogEntry`. Wie auch beim Event-Binding steht Ihnen hier die gesamte öffentliche API des jeweiligen DOM-Elements zur Verfügung. Wollten Sie beispielsweise zusätzlich den alternativen Text des Bildes mit dem Titel des Blog-Eintrags belegen, so wäre dies einfach über die folgende Anweisung möglich:

```
<img [src]="entry.image" [alt]="entry.title"/>
```

Details zu den unterschiedlichen Formen des Data-Bindings finden Sie ebenfalls in Kapitel 3, »Komponenten und Templating: der Angular-Sprachkern«. Die über das Formular eingegebenen Werte werden anschließend in der Listenansicht dargestellt.

Sie können nun also bereits damit beginnen, Ihren Blog, wie in Abbildung 1.10 dargestellt, mit Inhalt zu füllen.

Abbildung 1.10 Die laufende Blog-Anwendung

1.3.6 Modularisierung der Anwendung

Herzlichen Glückwunsch: Sie haben Ihre erste echte Angular-Anwendung fertiggestellt! Zum Abschluss dieser Einführung möchte ich Ihnen nun noch zeigen, wie Sie Ihre Applikation in einzelne Komponenten aufteilen und somit modularisieren können.

Die `AppComponent`-Komponente umfasst momentan sowohl den Quellcode zur Erzeugung der Einträge als auch das Markup für die Darstellung der Einträge in der Liste. In kleinen Applikationen ist dies gegebenenfalls auch in Ordnung; in größeren Applikationen werden Sie aber häufig in die Lage geraten, dass Sie bestimmte Komponenten an anderen Stellen Ihrer Anwendung wiederverwenden möchten oder dass Ihre Komponente schlichtweg zu groß und unübersichtlich wird.

In solchen Situationen spielt Angular eine seiner großen Stärken aus: So ist es mit sehr wenig Aufwand möglich, Teile einer Komponente herauszulösen und als eigenständige Komponente zu kapseln.

Im Folgenden werde ich Ihnen exemplarisch zeigen, wie Sie die Ansicht eines einzelnen Blog-Eintrags als eigenständige Komponente bereitstellen können.

Erstellen Sie hierfür zunächst die neue Datei *blog-entry.component.ts*, und definieren Sie dort die Komponente `BlogEntryComponent`:

```
import {Component, Input} from '@angular/core';
import {BlogEntry} from './blog-entry';
@Component({
  selector: 'app-blog-entry',
  templateUrl: './blog-entry.component.html'
})
export class BlogEntryComponent {
  @Input() entry?: BlogEntry;
}
```

Listing 1.14 »blog-entry.component.ts«: Die neue Klasse »BlogEntryComponent«

Best Practices für die Strukturierung von Angular-Projekten und das Angular-CLI

Bei der Strukturierung von Angular-Projekten hat es sich als gute Praxis bewährt, die Dateien zur Definition einer Komponente in einem eigenen Unterordner zusammenzufassen. So sollten Sie die Dateien *blog-entry.component.ts* und *blog-entry.component.html* beispielsweise im Ordner *blog-entry* speichern.

Im nächsten Kapitel werden Sie in diesem Zusammenhang sehen, wie das Angular-CLI Sie bei der Einhaltung solcher Regeln unterstützt. So stehen Ihnen unter anderem diverse *Generatoren* für die Erzeugung von Komponenten, NgModules und weiteren Angular-Sprachkonstrukten zur Verfügung.

In Listing 1.14 sehen Sie wieder zwei neue Angular- bzw. TypeScript-Features in Aktion: die Verwendung des `@Input`-Decorators sowie die Definition einer *optionalen Eigenschaft*. Konzentrieren Sie sich zunächst auf den `@Input`-Decorator: Mithilfe dieses Decorators können Sie die Eingangsparameter – und somit *die Schnittstelle* – Ihrer neuen Komponente definieren. (Den ebenfalls vorhandenen `@Output`-Decorator zur Deklaration der ausgehenden Events lernen Sie ebenfalls in Kapitel 3, »Komponenten und Templating: der Angular-Sprachkern«, kennen.)

Indem Sie ein Fragezeichen an den Namen der Member-Variablen hängen (`entry?`), sagen Sie TypeScript in diesem Zusammenhang, dass es möglich ist, dass diese Variable keinen Wert enthält – also den Wert `undefined` hat. Bei der weiteren Arbeit mit dieser Variablen müssen Sie diesen Fall dann explizit behandeln.

Kurzexkurs: Optional-Properties, Optional-Chaining und Strict-Property-Initialization – sicheres Behandeln von nicht definierten Variablen

Insbesondere bei der Verwendung von Input-Bindings kann es sehr leicht vorkommen, dass eine Variable keinen Wert von außen übergeben bekommt. In diesem

Zusammenhang hilft Ihnen das TypeScript-Sprachfeature *Strict-Property-Initialization*, den `undefined`-Fall sicher zu behandeln. So haben Sie in Listing 1.14 gesehen, wie Sie über das Fragezeichen ein optionales Property definieren können. Möchten Sie nun im weiteren Verlauf auf die `entry`-Variable zugreifen, so wird der TypeScript-Compiler Sie dazu zwingen, sicherzustellen, dass die Variable einen Wert enthält (siehe Abbildung 1.11).

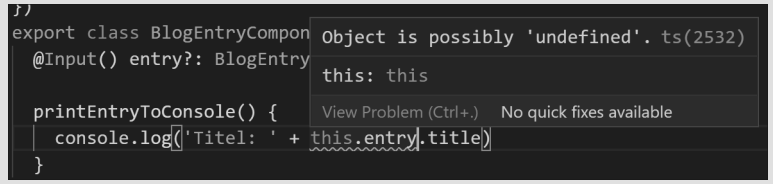

Abbildung 1.11 Fehlerhafter Zugriff auf ein optionales Property

Um diese Situation zu lösen, haben Sie diverse Möglichkeiten:

1. Sie weisen der `entry`-Variablen bei der Erzeugung einen Default-Wert zu und können so auf das Fragezeichen verzichten:

   ```
   @Input entry = new BlogEntry();
   ```

2. Sie Sie stellen über ein `if`-Statement sicher, dass die Variable einen Wert enthält:

   ```
   if(this.entry){
     console.log('Titel:' + this.entry.title);
   }
   ```

3. Sie verwenden *Optional-Chaining*, um nur auf das `title`-Property zuzugreifen, wenn die `entry`-Variable definiert ist:

   ```
   printEntryToConsole2() {
     console.log('Titel:' + this.entry?.title);
   }
   ```

   Das Fragezeichen beim Zugriff auf die Eigenschaft besagt an dieser Stelle, dass der Ausdruck nur dann weiter ausgewertet werden soll, wenn die `entry`-Eigenschaft einen gültigen Wert (also nicht `undefined` und nicht `null`) enthält.

Sie werden alle diese Techniken im weiteren Verlauf des Buches noch im Detail anwenden. Des Weiteren finden Sie in Abschnitt B.8, »null- und undefined-Handling: Arbeit mit optionalen Werten und Eigenschaften«, eine dedizierte Zusammenfassung des Themas. Für den Moment sollten Sie aber mitnehmen, dass TypeScript Ihnen umfangreiche Unterstützung bei der Behandlung von `undefined`- und `null`-Werten bietet!

Erstellen Sie nun die Datei *blog-entry.component.html*, und kopieren Sie das Markup, das den einzelnen Blog-Eintrag beschreibt, aus der Datei *app.component.html* in diese Datei:

```
<div class="blog-entry">
  <div class="blog-image">
    <img [src]="entry?.image" [alt]="entry?.title"/>
  </div>
  <div class="blog-summary">
    <span class="title">{{entry?.title}}</span>
    <p> {{entry?.text}}</p>
  </div>
</div>
```

Listing 1.15 »blog-entry.component.html«: ausgelagertes Template für die Darstellung eines Blog-Eintrags

Optional-Chaining in Angular-Templates

Beachten Sie auch hier, dass der Zugriff auf die Eigenschaften des `entry`-Objekts über den *Optional-Chaining*-Operator erfolgt (`entry?.image`). Da Sie das `entry`-Property in Ihrem TypeScript-Code als *optional* markiert haben, müssen Sie den `undefined`-Fall auch beim Zugriff aus dem Template heraus behandeln! Im Kontext von Angular-Templates wird der Optional-Chaining-Operator im übrigen auch *Safe-Navigation*-Operator genannt.

Achten Sie an dieser Stelle darauf, die Definition der `ngFor`-Schleife nicht zu übernehmen.

Fertig! Sie haben eine gekapselte Komponente entwickelt. Jetzt müssen Sie nur noch diese Komponente in der Anwendung bekannt machen und in der `AppComponent` verwenden. Erweitern Sie hierfür einfach das `declarations`-Array Ihres Applikationsmoduls um die neue Komponente:

```
import {AppComponent} from './app.component';
import {BlogEntryComponent} from './blog-entry/blog-entry.component';

@NgModule({
  imports: [BrowserModule],
  declarations: [AppComponent, BlogEntryComponent],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Listing 1.16 »app.module.ts«: Die »BlogEntryComponent« bei der Applikation bekannt machen

Die `BlogEntryComponent` steht Ihnen nun in allen Komponenten der Anwendung zur Verfügung, sodass Sie sie ab jetzt im Template der `AppComponent` verwenden können:


```
<app-blog-entry *ngFor="let entry of entries" [entry]="entry">
</app-blog-entry>
```

Listing 1.17 »app.component.html«: die »BlogEntryComponent« verwenden

Wie im `@Component`-Decorator der `BlogEntryComponent` festgelegt, erfolgt die Instanziierung der Komponente dabei über das Tag `<app-blog-entry>`. Da Sie eine Liste dieser Einträge erzeugen wollen, können Sie die `ngFor`-Schleife direkt innerhalb des Tags definieren.

Über das Property-Binding

```
[entry]="entry"
```

binden Sie schließlich die in der Komponentenkonfiguration deklarierte `@Input`-Variable an die lokale Zählvariable der Schleife. Wie Sie sehen, erfolgt dieses Binding mithilfe der gleichen Syntax, mit der Sie auch die öffentliche Schnittstelle von Standard-HTML-Elementen ansprechen.

1.4 Zusammenfassung und Ausblick

Vermutlich raucht Ihnen jetzt der Kopf. Sie haben in diesem Kapitel gelernt, wie Sie mithilfe von TypeScript Sprachelemente wie Klassen, Module und Typen verwenden können, aus welchen Bestandteilen eine Angular-Anwendung besteht und wie Sie Ihre Applikation in gekapselte Komponenten zerlegen können. Die folgende Liste fasst noch einmal die wichtigsten Erkenntnisse dieses Kapitels zusammen:

- TypeScript erweitert den ECMAScript-Standard zusätzlich um die Möglichkeit, statische Typen zu definieren sowie Metadaten in Form von Decorators zu erzeugen.
- Mithilfe der ECMAScript-Modulsyntax können Sie gekapselte Module definieren und diese bei Bedarf in Ihre Anwendung importieren.
- Angular-Komponenten bestehen aus einer TypeScript-Klasse sowie aus einem zugehörigen HTML-Template.
- Die Konfiguration einer Angular-Komponente erfolgt über den `@Component`-Decorator.
- Innerhalb eines Templates können Sie mithilfe des #-Zeichens lokale Variablen definieren.
- Mithilfe von Property-Bindings können Sie in Ihrem Template auf Variablen aus Ihrer TypeScript-Klasse zugreifen und diese an Elemente im DOM-Baum binden.
- Property-Bindings werden durch eckige Klammern `[ ]` definiert.
- Interpolationen sind eine vereinfachte Form des Property-Bindings. Sie werden durch doppelte geschweifte Klammern `{{ }}` definiert.

- Event-Bindings erlauben es Ihnen, auf DOM-Events wie `click` oder `keyup` zu reagieren und in diesem Fall beispielsweise Methoden der TypeScript-Klasse aufzurufen. Sie ermöglichen es Ihnen somit, Daten aus der View in die Komponenten-Klasse zu übertragen.
- Event-Bindings werden über runde Klammern `( )` definiert.
- Besitzt Ihre Komponente Eingangsparameter, so können Sie diese mithilfe des `@Input`-Decorators auszeichnen.
- Bei der Verwendung der Komponente können Sie über diese Eingangsparameter – wie im Property-Binding – mithilfe von eckigen Klammern `[ ]` Werte an Ihre Komponente übergeben.
- TypeScript bietet Ihnen umfangreiche Unterstützung für die Behandlung von `undefined`- und `null`-Werten an.
- Der Start einer Applikation erfolgt durch die Definition eines `NgModule` und die Übergabe dieses Moduls an die verwendete Plattform.
- Die `declarations`-Eigenschaft von `NgModule` ermöglicht es, Komponenten für das gesamte Modul bekannt zu machen.

Im folgenden Kapitel stelle ich Ihnen das Angular-Command-Line-Interface vor, bevor Sie dann in den kommenden Kapiteln in die Tiefen der Angular-Entwicklung eintauchen werden.

Kapitel 5

Fortgeschrittene Komponenten-Konzepte

Nachdem Sie nun die Grundlagen der Angular-Entwicklung kennen, geht es in diesem Kapitel in die Tiefe. Das Framework wartet mit einer Reihe interessanter Innovationen auf, die Ihnen bei der Entwicklung von professionellen Webanwendungen behilflich sein werden.

Mit den Grundlagen, die Sie bereits gelernt haben, können Sie tiefer in alle weiteren Bereiche der Angular-Entwicklung einsteigen. Neben den vorgestellten Basiskonzepten bietet Ihnen das Framework zusätzlich noch einige fortgeschrittene Komponenten-Konzepte und Techniken, die Ihnen bei der Entwicklung von komplexen Webanwendungen durchaus behilflich sein können. So werden Sie in diesem Kapitel unter anderem lernen,

- welche Möglichkeiten Ihnen Angular bietet, um das CSS-Styling Ihrer Komponenten direkt an den Komponenten zu pflegen.
- wie Sie mithilfe der Klassen `TemplateRef` und `NgTemplateOutlet` Teile Ihres Komponenten-Markups dynamisch austauschen können.
- wie Sie die Klasse `ViewContainerRef` dazu verwenden können, Komponenten aus Ihrem Komponenten-Code heraus zu instanziieren und zu löschen.
- wie der Angular-ChangeDetection-Mechanismus funktioniert und welche Optionen Ihnen zur Optimierung des ChangeDetection-Verhaltens und somit zur Optimierung Ihrer Performance zur Verfügung stehen.

5.1 Styling von Angular-Komponenten

Mit CSS3 steht Ihnen eine sehr ausdrucksstarke Sprache zur Gestaltung Ihrer Webanwendungen zu Verfügung. Sollten Sie bereits an größeren Webprojekten mitgearbeitet haben, so werden Sie aber mit Sicherheit auch wissen, dass CSS durchaus seine Tücken hat. Insbesondere die Tatsache, dass CSS-Regeln global definiert werden – und sich somit auf die gesamte Webapplikation auswirken –, führt in diesem Zusammenhang immer wieder zu Problemen. So hat vermutlich jeder Webentwickler schon einmal erlebt, dass eine Regel, die er nachträglich für eine neue Komponente

eingeführt hat, dazu führte, dass an einer anderen Stelle der Anwendung das Layout zerstört wurde (und das oft zunächst unbemerkt). CSS-Präprozessoren wie Sass oder Less unterstützen Entwickler in diesem Zusammenhang zwar stark – das Kernproblem bleibt aber auch bei der Nutzung dieser Tools bestehen: Definierte Regeln gelten immer für die gesamte Applikation.

5.1.1 Styles an der Komponente definieren

Angular bietet für diese Problematik einen sehr eleganten Lösungsansatz. So ist es möglich, über die beiden Eigenschaften `styles` und `styleUrls` des `@Component`-Decorators CSS-Regeln direkt an einer Komponente zu hinterlegen.

Die style-Eigenschaft

Listing 5.1 zeigt exemplarisch die Implementierung einer sehr einfachen `RedCircle`-Komponente:

```
@Component({
  selector: 'ch-red-circle',
  template: '<div></div>',
  styles: [`
     div {
       border-radius: 50%;
       width: 40px;
       height: 40px;
       background-color: red;
     }
  `]
})
export class RedCircleComponent {
}
```

Listing 5.1 »red-circle.ts«: Definition von Styles direkt an der Komponente

Der einzige Zweck dieser Komponente besteht darin, einen roten Kreis zu zeichnen. Auffällig ist hierbei zunächst, dass das implementierte Template komplett auf die Verwendung von CSS-Klassen verzichtet:

```
template: '<div></div>'
```

In diesem Zusammenhang ist es wichtig zu wissen, dass der *Angular-Styling-Mechanismus* die Auswirkungen der an einer Komponente hinterlegten CSS-Regeln auf diese beschränkt, sodass es oft nicht notwendig ist, innerhalb des HTML-Codes CSS-Klassen zu verwenden. Details hierzu erläutere ich in Abschnitt 5.1.2. Über die `styles`-Eigenschaft werden anschließend die benötigten Regeln definiert.

Alternativ ist es an dieser Stelle ebenfalls möglich, über die Eigenschaft `styleUrls` auf eine oder mehrere CSS-Dateien zu verweisen. Diese Variante werden Sie im weiteren Verlauf des Kapitels ebenfalls noch in Aktion sehen. Der Verwendung der Komponente erfolgt auf altbekannte Weise:

```
<ch-red-circle></ch-red-circle>
<div>Ich bin ein normales div-Tag</div>
```

Listing 5.2 »app.component.html«: Einsatz der »RedCircle«-Komponente

Wie erwartet, wird als Ergebnis lediglich die `RedCircleComponent` mit den definierten Regeln gestylt. Das zusätzliche `div`-Tag bleibt, wie Sie in Abbildung 5.1 sehen, vom CSS unberührt.

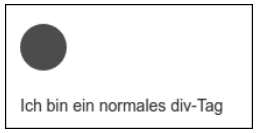

Abbildung 5.1 Darstellung des roten Kreises

5.1.2 ViewEncapsulation – Strategien zum Kapseln Ihrer Styles

Doch wie genau schafft es Angular jetzt eigentlich, dass die hinterlegten Regeln nur auf die jeweilige Komponente angewendet werden? Der Schlüssel zu diesem Rätsel liegt in den von Angular bereitgestellten *View-Encapsulation-Strategien*. Diese Strategien werden am `@Component`-Decorator definiert und legen fest, inwiefern das implementierte CSS für die Komponente gekapselt werden soll. Insgesamt bietet das Framework Ihnen drei Optionen zur Auswahl:

- `ViewEncapsulation.Emulated`: Die Style-Kapselung wird durch Angular emuliert. (Dies ist der Standardwert.)
- `ViewEncapsulation.None`: Es findet keine Kapselung der hinterlegten Styles statt.
- `ViewEncapsulation.ShadowDom`: Die Kapselung erfolgt über den nativen *WebComponents-Shadow*-DOM-Mechanismus.

Welche Strategie zum Einsatz kommt, legen Sie in der `encapsulation`-Eigenschaft des `@Component`-Decorators fest:

```
import {Component, ViewEncapsulation} from '@angular/core';
@Component({
   ...
  encapsulation: ViewEncapsulation.ShadowDom
})
export class MyComponent {
...
}
```

Der Standardfall: ViewEncapsulation.Emulated

Entscheiden Sie sich nicht explizit für eine Strategie, so wird automatisch die `ViewEncapsulation.Emulated`-Strategie für Ihre Komponente aktiviert. Diese sorgt, wie ich bereits kurz angedeutet habe, dafür, dass Angular die Kapselung der Styles für den Browser emuliert.

Was dies konkret bedeutet, sehen Sie am besten bei einem Blick in die Developer-Konsole. Schauen Sie sich hierfür zunächst das `<head>`-Tag der gestarteten Anwendung an. Neben den von Ihnen selbst definierten Einträgen werden Sie dort eine Reihe von `<style>`-Tags entdecken, die Angular generiert hat.

Jedes einzelne Tag enthält die CSS-Styles für eine bestimmte Komponente. Wenn Sie sich durch die Tags klicken, werden Sie auch auf den Block treffen, der für die `RedCircleComponent` zuständig ist (siehe Abbildung 5.2).

```
▼<style>
   div[_ngcontent-lou-14] {
          border-radius: 50%;
          width: 40px;
          height: 40px;
          background-color: red;
        }
  </style>
```

Abbildung 5.2 Von Angular generierte CSS-Definition

Wie Sie sehen, hat Angular aus dem einfachen `div`-Selektor den Selektor

```
div[_ngcontent-lou-14] {
    ...
}
```

generiert. Der Selektor trifft somit nur noch auf `div`-Tags zu, die zusätzlich das Attribut `_ngcontent-lou-14` besitzen. Untersuchen Sie nun mithilfe der Developer-Konsole den weiteren Quelltext der gerenderten Applikation, werden Sie feststellen, dass Angular exakt dieses Attribut ebenfalls zum `div`-Tag der `RedCircleComponent` hinzugefügt hat. Das `div`-Tag außerhalb der Komponente wurde hingegen nicht manipuliert. Abbildung 5.3 zeigt das entsprechend gerenderte Markup.

```
▼<ch-red-circle _nghost-lou-14>
   <div _ngcontent-lou-14></div>
  </ch-red-circle>
  <div>Ich bin ein normaler div-Tag</div>
```

Abbildung 5.3 Generierter HTML-Code der gerenderten Komponente

So schafft es Angular durch einen recht einfachen Trick, die Auswirkungen von CSS-Regeln auf die aktuelle Komponente zu beschränken.

Kapselung von CSS? Ja, aber nicht zu viel!

Die Möglichkeit, CSS-Styles direkt an der Komponente zu hinterlegen, ist durchaus verlockend: Anstatt sich für jedes Styling neue Klassennamen ausdenken zu müssen, können Sie so sicher sein, dass Ihre auf Komponenten-Ebene implementierten Regeln wirklich auch nur dort Auswirkungen haben.

Lassen Sie sich deswegen aber nicht dazu hinreißen, sämtliche CSS-Regeln nur noch in Ihren Komponenten zu definieren: Dies kann Ihnen am Ende mehr Arbeit machen, als es Ihnen nützt. So kann beispielsweise die Festlegung von `margin`- und `padding`-Werten auf Komponenten-Ebene schnell dazu führen, dass Sie für eine Änderung Ihres Themes jede einzelne Komponente der Anwendung anfassen müssen.

Mein Tipp lautet hier: Ist ein CSS-Style für das Aussehen der Komponente ausschlaggebend, so sollten Sie ihn auch direkt mit dieser Komponente kapseln. Handelt es sich jedoch um Dinge wie die Festlegung von Abständen innerhalb der Seite oder Ähnliches, gehören diese Regeln eher in ein globales Stylesheet. Bei einem späteren Refactoring werden Sie sich über die fehlende Kapselung sicher freuen!

::ng-deep: Aus der ViewEnapsulation.Emulated ausbrechen

Insbesondere dann, wenn Sie fremde Komponenten verwenden, kann es immer wieder einmal vorkommen, dass die standardmäßig vorhandene Kapselung der Styles zu Problemen führt. Stellen Sie sich beispielsweise vor, ein Fremdentwickler hat die für Sie perfekt passende Uploader-Komponente entwickelt. Leider hat dieser Entwickler aber alle Buttons der Komponente in Blau eingefärbt, obwohl Ihre Anwendung ansonsten in Grüntönen gehalten ist. Listing 5.3 zeigt die entsprechende Regel im Code der Uploader-Komponente:

```
.awsm-btn {
    color: #eeeeee;
    background-color: #337ab7;
}
```

Listing 5.3 »awesome-uploader.component.css«:
Definition des Button-Styles der Uploader-Komponente

Verwendet Ihre eigene Komponente nun die Style-Encapsulation »`Emulated`«, ist es zunächst einmal nicht ohne Weiteres möglich, die Regeln der fremden Komponenten zu überschreiben. So würde die folgende Regel ohne weitere Anpassungen dazu führen, dass Angular den CSS-Selektor mit einem dynamischen Attributselektor versieht, der anschließend verhindert, dass die Regel für die fremde Komponente »greift«:

```
.awsm-btn {
    background-color: forestgreen !important;
}
```

Listing 5.4 »styling-demo.component.html«: erfolgloser Versuch, die Komponenten-Styles aus Ihrer Komponente heraus zu überschreiben

Um dieses Problem zu beheben, stellt Angular Ihnen mit dem Pseudo-Selektor `::ng-deep` eine Möglichkeit zur Verfügung, die Generierung des dynamischen Attribut-Selektors zu deaktivieren. Listing 5.5 zeigt die entsprechende Anwendung in Ihrer eigenen Komponente:

```
::ng-deep .awsm-btn {
    background-color: forestgreen !important;
}
```

Listing 5.5 »styling-demo.component.html«: Einsatz des »::ng-deep«-Pseudo-Selektors zum Überschreiben der Button-Farbe

Ein Blick in die Oberfläche zeigt, dass der Button nun, wie gewünscht, in Grün dargestellt wird (siehe Abbildung 5.4).

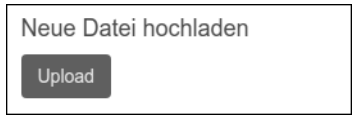

Abbildung 5.4 Darstellung des Buttons in Grün

Schauen Sie sich in diesem Zusammenhang auch ruhig noch einmal die Styles des Buttons über die Developer-Konsole an (siehe Abbildung 5.5).

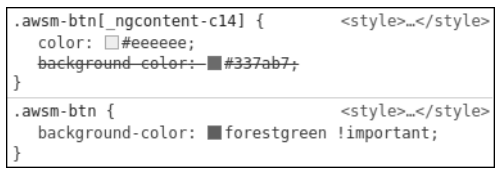

Abbildung 5.5 Ansicht der Styles in der Developer-Konsole

Wie Sie sehen, enthält die Regel, die den Button grün einfärbt, wie erwartet keinen Attribut-Selektor und wird somit – unabhängig vom aktuellen Kontext – auf alle Elemente der Seite angewendet, die die Klasse `awsm-btn` besitzen.

Deprecation von ::ng-deep

Zu dem Zeitpunkt, als ich dieses Buch schrieb, war der `::ng-deep`-Selektor in der Angular-Dokumentation als »deprecated«, also als »veraltet« markiert. Um diese

Kennzeichnung gibt es seitdem einige Diskussionen in der Angular-Community. Der Grund für die Deprecation besteht in diesem Zusammenhang darin, dass der `/deep/`-Selektor, der das gleiche Verhalten für den Shadow-DOM-Standard definieren sollte, mittlerweile aus diesem Standard entfernt wurde. Laut Angular-Team-Mitgliedern gibt es aber zum aktuellen Zeitpunkt weder einen äquivalenten Ersatz für den `::ng-deep`-Selektor noch konkrete Überlegungen, diesen wirklich aus Angular zu entfernen. Sie können den Selektor also – trotz Deprecation – guten Gewissens weiterhin verwenden.

CSS-Kapselung abschalten: ViewEncapsulation.None

In gewissen Fällen ist es sinnvoll, die Kapselung von CSS-Styles abzuschalten. Ein typisches Beispiel hierfür ist die Implementierung von Container-Komponenten. Stellen Sie sich beispielsweise vor, Sie wollten eine speziell gestylte Dialog-Komponente entwerfen. Der Benutzer Ihrer Komponente soll ebenfalls die von Ihnen bereitgestellten Klassen verwenden können, um dort beispielsweise passend gestaltete Buttons zu platzieren. Die Benutzung dieser Komponente könnte etwa wie folgt aussehen:

```
<ch-styled-dialog title="Wollen Sie Ihre Styles kapseln?">
  <button class="ch-styled-dialog-button">Ja</button>
  <button class="ch-styled-dialog-button">Nein</button>
</ch-styled-dialog>
```

Listing 5.6 Verwendung des gestylten Dialogs

Mit der Standardstrategie `Emulated` wären die für den `StyledDialog` definierten CSS-Regeln im Body der Komponente aber überhaupt nicht sichtbar. Die Klasse `ch-styled-dialog-button` würde somit ohne Auswirkung bleiben. Möchten Sie die innerhalb der Komponente definierten Styles auch den Kind-Elementen zugänglich machen, so verwenden Sie `StyleEncapsulation.None`. Listing 5.7 zeigt die entsprechende Komponenten-Implementierung:

```
import {Component, Input, ViewEncapsulation} from '@angular/core';
@Component({
  selector: 'ch-styled-dialog',
  templateUrl: 'styled-dialog.component.html',
  styleUrls: ['styled-dialog.component.css'],
  encapsulation: ViewEncapsulation.None
})
export class StyledDialogComponent {
  @Input() title?: string;
}
```

Listing 5.7 »styled-dialog.component.ts«: Einsatz der »ViewEncapsulation.None«-Strategie

Im `@Component`-Decorator sehen Sie einerseits die Angabe der `Encapsulation`-Strategie sowie andererseits die Verwendung der `styleUrls`-Eigenschaft zum Verweis auf eine CSS-Datei. Innerhalb dieser Datei sind nun alle Regeln definiert, die sowohl für die Komponente selbst als auch für alle anderen Elemente der Applikation gültig sein sollen:

```
.ch-styled-dialog {
    background-color: #fff;
    border: 1px solid #ddd;
}
...
.ch-styled-dialog-button {
    display: inline-block;
    background-color: white;
    padding: 4px 12px;
    ...
}
```

Listing 5.8 »styled-dialog.component.css«: Auszug aus der CSS-Datei

Durch die Verwendung der `ViewEncapsulation.None`-Strategie werden die Buttons nun korrekt dargestellt, wie Sie in Abbildung 5.6 sehen.

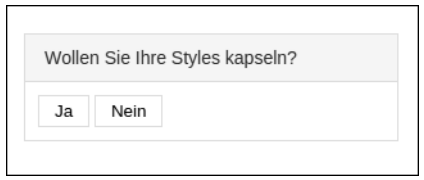

Abbildung 5.6 Gerenderte Dialog-Komponente

Auch die Untersuchung des Quellcodes zeigt, dass weder die generierten CSS-Regeln im Kopf der Seite noch der erzeugte HTML-Code für die Komponente von Angular manipuliert wurden (siehe Abbildung 5.7 und Abbildung 5.8).

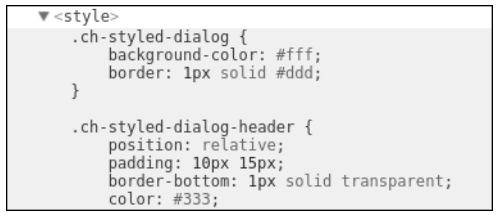

Abbildung 5.7 Generierter CSS-Code

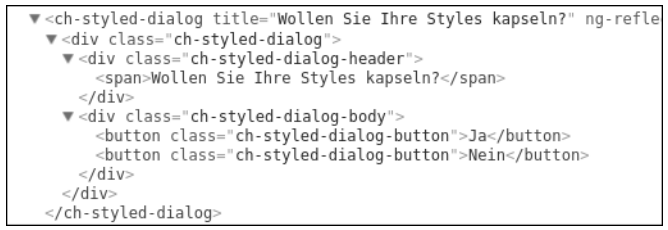

Abbildung 5.8 HTML-Code bei Verwendung der Komponente

Auswirkung von »ViewEncapsulation.None« auf andere Elemente der Anwendung

Beachten Sie an dieser Stelle, dass die innerhalb der Komponente definierten CSS-Regeln durch die Abschaltung der ViewEncapsulation nun natürlich auch Auswirkung auf alle anderen Elemente der gerenderten Seite haben. Hätten Sie innerhalb Ihrer Komponente beispielsweise die Regel

```
div {
  background-color: blue;
}
```

definiert, würden nun plötzlich alle `<div>`-Elemente der Anwendung eine blaue Hintergrundfarbe erhalten. Sollten Sie sich für `ViewEncapsulation.None` entscheiden, empfehle ich Ihnen, über eindeutige CSS-Klassennamen sicherzustellen, dass Ihre Regeln keine unerwünschten Nebeneffekte haben. Eine Technik, die Sie hierbei unterstützen kann, ist die *Block-Element-Modifier*-Technik (BEM), die ich Ihnen im Rahmen der Entwicklung von eigenen Angular-Bibliotheken in Kapitel 19 im Detail vorstelle.

Vollständige Kapselung: ViewEncapsulation.ShadowDom

Die letzte Möglichkeit zum Kapseln von CSS-Regeln ist die `ViewEncapsulation.ShadowDom`-Strategie. Diese Strategie verlässt sich (wie der Name bereits vermuten lässt) auf den nativen Kapselmechanismus des *Shadow-DOM* – einer Funktionalität aus dem *WebComponents*-Standard.

Die Grundidee hinter der Shadow-DOM-Technik besteht vereinfacht ausgedrückt darin, bestimmte Teile des DOM-Baums vor der restlichen Applikation »zu verstecken«. So ist die interne Implementierung einer Komponente im Optimalfall völlig irrelevant für die Anwendung, die die Komponente verwendet, und sollte somit auch nicht für diese sichtbar (und veränderbar) sein.

Diese Anforderung löst die Shadow-DOM-Technik dadurch, dass genau solche internen Implementierungsdetails innerhalb eines sogenannten `shadow-root`-Knotens in die Applikation eingehängt werden. Auf alle Elemente innerhalb dieses Knotens

kann nun nicht mehr von außen zugegriffen werden; sie können also z. B. nicht mithilfe eines `querySelector`-Aufrufs über die DOM-API selektiert werden.

Des Weiteren garantiert das Shadow-DOM, dass keine CSS-Regeln, die innerhalb des `shadow-root`-Knotens definiert wurden, außerhalb des Knotens sichtbar sind und dass keine Regeln, die außerhalb definiert wurden, Auswirkungen auf das Styling der Komponente haben. Dieser letzte Punkt ist somit auch der für die Angular-Entwicklung relevante.

Während globale Styles bei der `ViewEncapsulation.Emulated`-Strategie in der Komponente sichtbar sind, bleiben sie bei der `ViewEncapsulation.ShadowDom`-Strategie ohne Effekt. Erinnern Sie sich hierfür noch einmal an die Implementierung der `TimepickerComponent`. Bei der Implementierung des Layouts haben Sie für die Aufteilung in drei Spalten die Klasse `col` verwendet:

```
<div class="timepicker">
  <div class="col">
    <button (click)="incrementTime('hours')"> + </button>
    <input [value]="time?.hours" .../>
    <button (click)="decrementTime('hours')"> - </button>
  </div>
  ...
</div>
```

Durch die `ViewEncapsulation.Emulated`-Kapselung bleiben die für diese Klasse definierten Styles zwar ohne Auswirkung auf die restliche Anwendung; wenn allerdings in einem globalen CSS-Stylesheet ebenfalls ein Selektor für die Klasse `col` definiert wurde, werden diese Styles nun zusätzlich auf das `div`-Tag angewendet. Sollten Sie beispielsweise ein CSS-Framework verwenden, das zufällig die Klasse `col` wie folgt definiert

```
.col {
    background-color: lightgrey;
}
```

so würde dies dazu führen, dass Ihre Timepicker-Komponente urplötzlich die grau hinterlegten Flächen aufweist, die Sie in Abbildung 5.9 sehen.

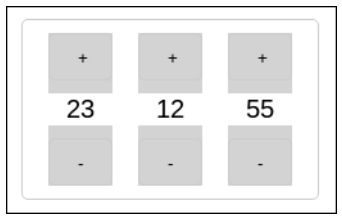

Abbildung 5.9 Timepicker-Komponente mit Styles aus dem globalen Stylesheet

Diesen Effekt können Sie verhindern, indem Sie die `ViewEncapsulation.ShadowDom`-Strategie verwenden. Listing 5.9 zeigt die entsprechende Konfiguration des Timepickers:

```
@Component({
  selector: 'time-picker',
  encapsulation: ViewEncapsulation.ShadowDom,
  ...
})
export class TimePickerComponent {
...
}
```

Listing 5.9 »time-picker.component.ts«: Konfiguration zur Verwendung des Shadow-DOM

Wie erwartet, wird der Timepicker anschließend wieder korrekt dargestellt.

Interessant ist in diesem Zusammenhang erneut der Blick in die Developer-Ansicht. Schauen Sie sich hierfür den generierten HTML-Code aus Abbildung 5.10 an.

Dieser Ausschnitt enthält einige sehr interessante Aspekte: Zunächst sehen Sie, dass – wie zuvor beschrieben – die internen Teile der Komponente unterhalb des `#shadow-root`-Knotens versteckt wurden. Des Weiteren wurden alle Styles des Timepickers direkt in diesen Knoten integriert, sodass sie nur innerhalb des Knotens sichtbar sind. Die Selektoren können aus diesem Grund ebenfalls unmanipuliert bleiben.

Auf den ersten Blick erscheint es etwas verwirrend, dass Angular zusätzlich zu den Styles der `TimePicker`-Komponente ebenfalls die Styles der anderen verfügbaren Komponenten in den `shadow-root`-Knoten kopiert.

Abbildung 5.10 »shadow-root«-Knoten zum Kapseln der Timepicker-Komponente

Der Grund hierfür ist, dass es andernfalls nicht möglich wäre, innerhalb einer nativ gekapselten Komponente andere Komponenten zu verwenden: Die native Kapselung würde in diesem Fall den Zugriff auf die Styles verhindern, die Sie im `head` des Dokuments definiert haben.

Auch wenn dieses Verhalten für die Praxistauglichkeit von Angular durchaus sinnvoll ist, können hierdurch dennoch unerwünschte Effekte entstehen: Sollten es neben Ihrer nativ gekapselten Komponente zusätzlich weitere Komponenten mit der Strategie `ViewEncapsulation.None` geben, so hätten die in diesen Komponenten definierten Styles nun doch wieder Einfluss auf die eigentlich abgekapselte Native-Komponente.

Shadow-DOM und Browserkompatibilität

Ein großer Schwachpunkt der Shadow-DOM-Technik ist, dass sie zum aktuellen Zeitpunkt (Januar 2022) noch nicht von allen Browsern unterstützt wird.

Sollten Sie also eine Anwendung schreiben, bei der Sie nicht die vollständige Kontrolle über die vom Nutzer verwendeten Browser haben, würde ich Ihnen momentan vom Einsatz der `ViewEncapsulation.ShadowDom`-Strategie abraten.

Eine sehr hilfreiche Adresse zur Überprüfung von Browserkompatibilität ist die Seite *caniuse.com*. So können Sie unter

http://caniuse.com/#feat=shadowdomv1

jederzeit verfolgen, welche Browser das Shadow-DOM bereits unterstützen und welche nicht.

5.2 TemplateRef und NgTemplateOutlet: dynamisches Austauschen von Komponenten-Templates

Bei der Entwicklung von wiederverwendbaren Komponenten ist es ein durchaus üblicher Anwendungsfall, dem Nutzer Ihrer Komponente die Möglichkeit zu geben, ihr Aussehen seinen Wünschen entsprechend anzupassen.

In den meisten Webanwendungen wird diese Anforderung dadurch erfüllt, dass man eigene CSS-Styles für die Komponenten bereitstellt, die gerendert werden sollen. Wie Sie in Abschnitt 5.1.2 bereits gesehen haben, ist dies in Angular ebenfalls leicht möglich, wenn Sie die `ViewEncapsulation.None`-Strategie verwenden. Angular geht bezüglich der Anpassbarkeit von Komponenten aber noch einen großen Schritt weiter.

Durch selbst definierte Template-Referenzen können Sie Entwicklern, die Ihre Komponenten-Bibliothek verwenden, sogar die Möglichkeit geben, Teile der Komponenten-View auszutauschen und durch eigene Templates zu ersetzen.

Erinnern Sie sich hierfür zunächst an das Blog-Beispiel aus Kapitel 1, »Angular-Kickstart: Ihre erste Angular-Webapplikation«. In dieser Anwendung wurde über ein Array von `Blog`-Objekten iteriert. Anschließend wurden die `Blog`-Objekte in der Liste gerendert, die Sie in Abbildung 5.11 sehen.

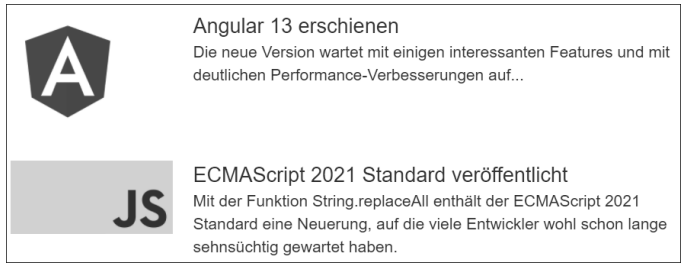

Abbildung 5.11 Darstellung der Blog-Einträge in der Liste

Stellen Sie sich nun vor, Sie möchten an anderer Stelle ebenfalls die Liste der Blog-Einträge darstellen, diesmal aber in einer minimalistischeren Form – zum Beispiel als Überblick in einer Sidebar. In diesem Fall wäre es schön, die Komponente wiederverwenden und lediglich die Darstellung der Listenelemente verändern zu können. Über die Klasse `TemplateRef` bietet Ihnen Angular exakt diese Möglichkeit.

5.2.1 NgFor mit angepassten Templates verwenden

In den Quelltexten dieses Kapitels finden Sie eine leicht abgewandelte Version der Blog-Anwendung aus Kapitel 1. In der neuen Version wurde die Listendarstellung zusätzlich in die Komponente `BlogListComponent` ausgelagert. Auf diesem Weg ist es in den kommenden Abschnitten leichter, die `TemplateRef`-Funktionalität verständlich zu erläutern.

Listing 5.10 zeigt als Ausgangspunkt für die weitere Implementierung zunächst, wie die Verwendung der `BlogListComponent` im »Normalfall« aussieht. In Listing 5.11 sehen Sie die Komponente bei Nutzung eines eigenen Templates.

```
<h2>Blogeinträge</h2>
<ch-blog-list [entries]="entries"></ch-blog-list>
```

Listing 5.10 »blog.component.html«: reguläre Verwendung der »BlogList«-Komponente

```
<ch-blog-list [entries]="entries">
  <ng-template let-entry>
    <div class="entry-minimal" >
      <div class="summary-image">
        <img [src]="entry?.image" [alt]="entry?.title" class="small"/>
      </div>
```

```
      <div class="summary-title">
        <span>{{entry?.title}}</span>
      </div>
    </div>
  </ng-template>
</ch-blog-list>
```

Listing 5.11 »blog.component.html«: Verwendung der »BlogListComponent« mit eigenem Template

Die Idee dahinter lautet wie folgt: »Wird kein eigenes Template mitgeliefert, soll die Standardansicht zum Rendern der Einträge verwendet werden. Wird ein Template mitgeliefert, soll dieses zur Darstellung herangezogen werden.«

Sie benötigen also innerhalb der `BlogListComponent` zunächst einmal Zugriff auf das im Body definierte `<ng-template>`. Hierfür stellt Angular die bereits angesprochene `TemplateRef`-Klasse bereit. In Verbindung mit dem bereits bekannten `@ContentChild`-Decorator können Sie so leicht in Ihrer Anwendungslogik auf das Template zugreifen:

```
@Component({
  selector: 'ch-blog-list',
  ...
})
export class BlogListComponent {
    @Input() entries: BlogEntry[] = [];
    @ContentChild(TemplateRef) entryTemplate?: TemplateRef<any>;
}
```

Listing 5.12 Implementierung der »BlogListComponent« und Zugriff auf das angepasste Template

Ist ein eigenes Template definiert, so können Sie nun im HTML-Code über die `entryTemplate`-Variable darauf zugreifen. Die tatsächliche Einbettung des Templates geschieht nun im HTML-Code der `BlogListComponent`:

```
<div *ngIf="!entryTemplate">
  <ch-blog-entry [entry]="entry" *ngFor="let entry of entries">
  </ch-blog-entry>
</div>
<div *ngIf="entryTemplate">
  <ng-template ngFor [ngForOf]="entries" [ngForTemplate]="entryTemplate">
  </ng-template>
</div>
```

Listing 5.13 »blog-list.component.html«: Verwendung des übergebenen Templates im HTML-Code

Der erste Block des Codes sollte Sie mittlerweile nicht mehr überraschen: Ist kein eigenes Template definiert, so wird mithilfe der `NgFor`-Direktive über die Einträge iteriert. Zur Darstellung der einzelnen Einträge wird die `BlogEntryComponent` verwendet. Im zweiten Teil wird es nun etwas spannender. Rufen Sie sich hierfür zunächst noch einmal die Funktionsweise der Angular-Microsyntax in Erinnerung: Die *-Schreibweise ist demnach lediglich eine Kurzschreibweise für die Arbeit mit HTML-Templates. So ist der Code

```
<ch-blog-entry [entry]="entry" *ngFor="let entry of entries">
</ch-blog-entry>
```

funktional identisch mit dieser Langschreibweise:

```
<ng-template ngFor let-entry [ngForOf]="entries">
  <ch-blog-entry [entry]="entry"></ch-blog-entry>
</ng-template>
```

Zusätzlich hierzu bietet die `NgFor`-Direktive Ihnen aber außerdem die Möglichkeit, über das `ngForTemplate`-Input-Binding ein zu renderndes Template an die Direktive zu übergeben. Über den Code

```
<ng-template ngFor [ngForOf]="entries" [ngForTemplate]="entryTemplate">
</ng-template>
```

teilen Sie `NgFor` somit mit, dass Sie für jedes Element der Schleife das `entryTemplate` rendern möchten. Eine etwas eigenwillige Besonderheit besteht zusätzlich darin, dass die Zählvariable `let-entry` in diesem Fall nicht im umschließenden `template`-Tag, sondern bei der Implementierung des `entryTemplate` definiert wird:

```
<ch-blog-list [entries]="entries">
    <ng-template let-entry>
        <div class="entry-summary" >
          ...
        </div>
    </ng-template>
</ch-blog-list>
```

Listing 5.14 Definition der Zählvariablen »let-entry« im »customTemplate«

Die neue Implementierung der `BlogListComponent` gibt den Benutzern der Komponente nun die volle Flexibilität in Hinblick auf die Darstellung der Listeneinträge. Das im Beispiel definierte `entryTemplate` führt beispielsweise dazu, dass Sie mit wenig Aufwand eine zusätzliche Kurzübersicht über Ihre vorhandenen Blog-Einträge bereitstellen können (siehe Abbildung 5.12).

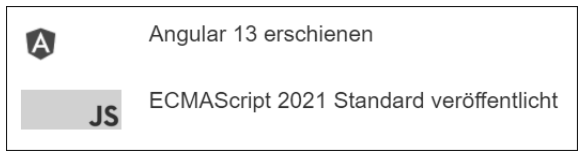

Abbildung 5.12 »BlogList« mit Custom-Renderer

5.2.2 NgTemplateOutlet: zusätzliche Templates an die Komponente übergeben

Sie wissen nun, wie Sie Templates dynamisch austauschen können, die von der `NgFor`-Direktive verwendet werden. In manchen Fällen kann es aber auch notwendig sein, Templates außerhalb von Schleifen hinzuzufügen oder zu verändern. Angular stellt Ihnen für diesen Anwendungsfall die `NgTemplateOutlet`-Direktive zur Verfügung. Sie verwenden sie recht ähnlich wie die zuvor vorgestellte `ngFor`-Syntax. Möchten Sie den Nutzern Ihrer `BlogListComponent` beispielsweise die Möglichkeit geben, zusätzliches Markup einzufügen (zum Beispiel zur Personalisierung der Komponente), so können Sie dies mithilfe der `NgTemplateOutlet`-Direktive wie in Listing 5.15 gezeigt realisieren:

```
<ng-template *ngIf="additionalMarkup"
             [ngTemplateOutlet]="additionalMarkup">
</ng-template>
```

Listing 5.15 »blog-list.component.html«: Verwendung der »NgTemplateOutlet«-Direktive

Bei der `additionalMarkup`-Variablen handelt es sich – genau wie bei der `entryTemplate`-Variablen – um ein `TemplateRef`-Objekt. Innerhalb Ihrer Komponente benötigen Sie nun also die Möglichkeit, zwei Template-Referenzen zu übergeben. Listing 5.16 zeigt, wie Sie die `BlogListComponent` in der Blog-Komponente verwenden:

```
<ch-blog-list [entries]="entries">
  <ng-template let-entry #entryTemplate>
    <div  class="entry-summary" >
       ...
    </div>
  </ng-template>
  <ng-template #additionalMarkup>
    <div>Insgesamt: {{entries.length}} Einträge</div>
  </ng-template>
</ch-blog-list>
```

Listing 5.16 »blog-component.html«: Übergabe von zwei Templates an die »BlogListComponent«

Beachten Sie in diesem Listing insbesondere die Definition der beiden lokalen Variablen `entryTemplate` und `additionalMarkup`. So haben Sie innerhalb der `BlogListComponent` über diese Variablen die Möglichkeit, die beiden Templates gezielt zu injizieren. Als Schlüssel dient dabei nicht mehr die Klasse des Elements (`TemplateRef`), sondern der Name der lokalen Variablen:

```
export class BlogListComponent {
    @Input() entries: BlogEntry[] = [];
    @ContentChild('entryTemplate') entryTemplate?: TemplateRef<any>;
    @ContentChild('additionalMarkup') additionalMarkup?: TemplateRef<any>;
}
```

Listing 5.17 »blog-list.component.ts«: Injektion der beiden Templates über den Namen der Template-Variablen

Die Nutzer haben nun die Möglichkeit, beliebiges Markup zu Ihrer Komponente hinzuzufügen (siehe Abbildung 5.13).

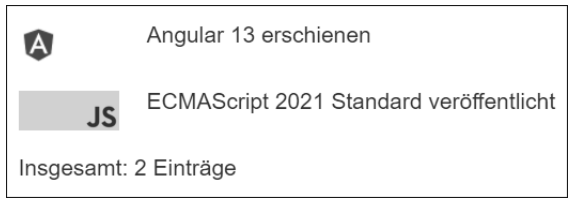

Abbildung 5.13 Darstellung der Blog-Liste nach dem Hinzufügen von zusätzlichem Markup

Des Weiteren haben Sie über die `NgTemplateOutlet`-Direktive die Möglichkeit, bestehende Templates dynamisch auszutauschen. Erinnern Sie sich hierfür noch einmal an das Formular zum Anlegen eines neuen Blog-Eintrags (siehe Abbildung 5.14).

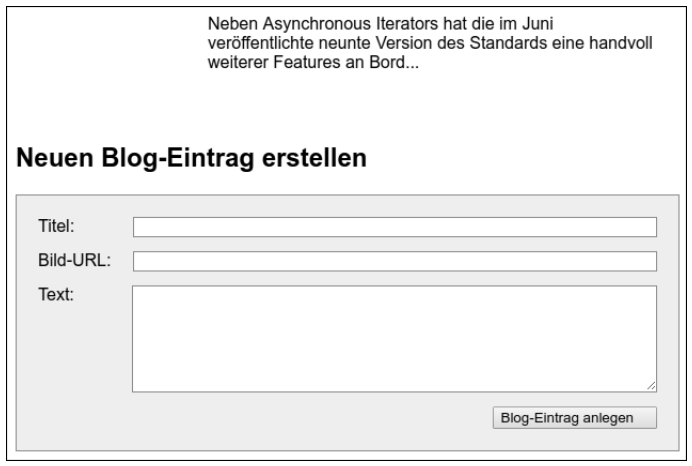

Abbildung 5.14 Standardansicht des Formulars

In den Quelltexten dieses Kapitels habe ich dieses Formular in eine eigene Komponente ausgelagert (*blog-form.component.ts*). Die Komponente kann bei Verwendung des Standard-Templates wie folgt eingesetzt werden:

```
<ch-blog-list [entries]="entries"></ch-blog-list>

<h2>Neuen Blog-Eintrag anlegen</h2>
<ch-blog-form (entryCreated)="saveEntry($event)"> </ch-blog-form>
```

Listing 5.18 »blog.component.html«: Verwendung der »BlogFormComponent« ohne eigenes Template

Möchten Sie den Nutzern Ihrer Komponente nun die Möglichkeit bieten, ein eigenes Formular-Layout bereitzustellen, so können Sie dies erneut durch die Übergabe eines Templates realisieren. Listing 5.19 zeigt die Definition eines eigenen Formular-Layouts, das direkt beim Anlegen eines Blog-Eintrags ein Vorschaubild des Blog-Images bereitstellt:

```
<ch-blog-form (entryCreated)="saveEntry($event)" #form>
  <ng-template>
    <div class="custom-form">
      <form>
        ...
        <form>
          ...
          <div class="right">
            <input type="url" #image placeholder="Bild"/>
            <div class="preview">
              <img [src]="image.value" >
            </div>
          </div>
          <div>
            <button (click)="form.createBlogEntry(title.value,
                                image.value, text.value)">
              Blog-Eintrag anlegen
            </button>
          </div>
        </form>
      </div>
    </ng-template>
</ch-blog-form>
```

Listing 5.19 »blog.component.html«: Verwendung der »BlogFormComponent« und Übergabe eines eigenen Templates zur Darstellung des Formulars

Wie schon bei den vorherigen Beispielen können Sie das Template nun mithilfe des `ContentChild`-Decorators in die `BlogFormComponent` injizieren:

```
export class BlogFormComponent {
  @Output() entryCreated = new EventEmitter();
  @ContentChild(TemplateRef) customTemplate?: TemplateRef;
  ...
}
```

Listing 5.20 »blog-form.component.ts«: Injektion des nutzerdefinierten Templates

Innerhalb des Templates können Sie anschließend mithilfe der `NgIf`-Direktive entscheiden, ob Sie die Standardansicht oder die vom Nutzer definierte Ansicht verwenden wollen:

```
<div class="form" *ngIf="!customTemplate">
  <div class="control">
    <label for="title">Titel:</label>
    <input type="text" id="title" #title/>
  </div>
  ...
</div>
<ng-template *ngIf="customTemplate" [ngTemplateOutlet]="customTemplate"></ng-
template>
```

Listing 5.21 »blog-form.component.html«: Verwendung der »customTemplate«-Variablen zur Darstellung der Standardansicht oder der nutzerdefinierten Ansicht

Wie Sie in Abbildung 5.15 sehen, wird das Formular nun mit dem neuen Layout in der Oberfläche dargestellt.

Abbildung 5.15 Darstellung des Formulars bei Übergabe eines eigenen Templates

5.3 ViewContainerRef: Komponenten zur Laufzeit hinzufügen

In allen bisherigen Beispielen wussten Sie bereits zur Entwicklungszeit, welche Komponenten Sie an welcher Stelle im Komponenten-Baum darstellen wollen. So war es in diesen Fällen leicht möglich, innerhalb eines Templates einfach das entsprechende Tag der Komponente zu verwenden.

Möchten Sie hingegen erst zur Laufzeit entscheiden, welche Komponente in den DOM-Tree eingefügt werden soll, stößt dieser statische Ansatz an seine Grenzen. Mithilfe der Klasse `ViewContainerRef` bietet Angular Ihnen jedoch zusätzlich die Möglichkeit, Komponenten zur Laufzeit der Anwendung zu einem Template hinzuzufügen.

In den folgenden Abschnitten zeige ich Ihnen am Beispiel einer generischen `CircleComponent`, wie Sie Komponenten dynamisch zu einem Template hinzufügen, dort verschieben und wieder löschen können. Des Weiteren werden Sie lernen, wie Sie über die Komponenten-Referenz mit der Komponente interagieren können, um dort beispielsweise aus Ihrem TypeScript-Code heraus Input-Bindings zu verändern.

Listing 5.22 zeigt zum Einstieg zunächst einmal die `CircleComponent` zur Darstellung eines Kreises. Über das Input-Binding `color` lässt sich hierbei die Hintergrundfarbe des Kreises verändern:

```
@Component({
  selector: 'ch-circle',
  template: `<div [ngStyle]="{'background-color' : color}"></div>`,
  styles: [...]
})
export class CircleComponent {
  @Input() color = 'black';
}
```

Listing 5.22 »dynamic-components-demo.component.ts«: Definition einer einfachen Circle-Komponente

5.3.1 ViewContainerRef: Komponenten zur Laufzeit hinzufügen

Möchten Sie diese Komponente nun dynamisch zu einer existierenden Komponente bzw. zu deren Template hinzufügen, benötigen Sie zunächst einmal einen »Ort« im Template (den sogenannten *ViewContainer*), an dem die Komponente(n) eingefügt werden soll(en). Erstellen Sie hierfür einfach ein beliebiges DOM-Element, und machen Sie dieses mithilfe einer lokalen Variablen (hier `container`) selektierbar:

```
<div class="element-container" #container></div>
```

Listing 5.23 »dynamic-components-demo.component.html«: Definition des »div«-Elements zum Einfügen der dynamischen Komponenten

Darauf aufbauend zeigt Listing 5.24 die Implementierung zur dynamischen Erzeugung von zwei Kreis-Komponenten:

```
export class DynamicComponentsDemoComponent implements AfterViewInit {
  @ViewChild('container', {read: ViewContainerRef}) container!: ViewContainerRef;
  ngAfterViewInit(){
    this.container.createComponent(CircleComponent); // Kreis 1
    this.container.createComponent(CircleComponent); // Kreis 2
  }
}
```

Listing 5.24 »dynamic-components-demo.component.ts«: dynamisches Einfügen von zwei Circle-Komponenten

Innerhalb der Komponenten-Klasse benötigen Sie zunächst eine Referenz auf den View-Container in Form der Klasse `ViewContainerRef`. Wie bei der Selektion von Kind-Komponenten erfolgt dies mithilfe des `@ViewChild`-Decorators. Als Schlüssel verwenden Sie dabei den Namen der lokalen Variablen. In Listing 5.25 sehen Sie nun aber außerdem eine neue Technik im Einsatz: die *read-Notation*.

```
export class DynamicComponentsDemoComponent implements AfterViewInit {
  @ViewChild('container', {read: ViewContainerRef}) container!:
      ViewContainerRef;
 ...
}
```

Listing 5.25 »dynamic-components-demo.component.ts«: Verwendung der »read«-Notation zum Zugriff auf den »ViewContainer«

So kann das selektierte DOM-Element je nach Einsatzzweck unterschiedliche »Rollen« einnehmen. Möchten Sie beispielsweise Zugriff auf das Native-DOM-Element erhalten, so benötigen Sie dafür eine Instanz der Klasse `ElementRef` (siehe auch Kapitel 4, »Direktiven: Komponenten ohne eigenes Template«). Möchten Sie hingegen auf ein HTML5-Template zugreifen, erfolgt dies über die Klasse `TemplateRef` (siehe Abschnitt 5.2), und für die Arbeit mit dynamischen Templates benötigen Sie die `ViewContainerRef`. Mithilfe der `read`-Notation haben Sie hier die Möglichkeit, je nach Einsatzzweck explizit zu bestimmen, welche Referenz Sie gerade benötigen.

Innerhalb des `ngAfterViewInit`-Callbacks können Sie nun die `createComponent`-Methode der `ViewContainerRef`-Klasse verwenden, um neue Komponenten in den Komponenten-Baum einzufügen:

```
ngAfterViewInit(){
  this.container.createComponent(CircleComponent);
  this.container.createComponent(CircleComponent);
}
```

Ein erneuter Blick auf die Oberfläche zeigt, dass nun tatsächlich zwei `CircleComponent`-Instanzen im View der `DynamicComponentsDemoComponent` dargestellt werden (siehe Abbildung 5.16).

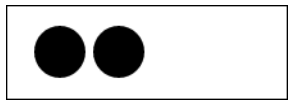

Abbildung 5.16 Darstellung der beiden dynamisch hinzugefügten Kreise

5.3.2 ComponentRef: Interaktion mit der dynamisch erzeugten Komponente

So weit, so gut: Sie wissen nun, wie Sie Komponenten über die Methode `createComponent` zur Laufzeit zu einem Template hinzufügen können. Um echte Anwendungsfälle mit dem Mechanismus abdecken zu können, benötigen Sie aber selbstverständlich die Möglichkeit, mit der erzeugten Komponente auch nach ihrer Erzeugung zu interagieren.

Für diesen Zweck liefert die `createComponent`-Methode Ihnen als Rückgabewert eine Instanz der Klasse `ComponentRef`. Über deren Eigenschaft `instance` haben Sie anschließend Zugriff auf die erzeugte Komponenten-Instanz. Möchten Sie beispielsweise das Input-Binding `color` nach der Erzeugung der Komponente verändern, so können Sie dies wie folgt erreichen:

```
export class DynamicComponentsDemoComponent implements AfterViewInit {
  ...
  addCircle(color: string) {
    const circleRef = this.container.createComponent(CircleComponent);
    circleRef.instance.color = color;
  }
}
```

Listing 5.26 »dynamic-components-demo.component.ts«: Verwendung der »ComponentRef«-Instanz zum Setzen des Input-Bindings

Listing 5.27 zeigt, wie Sie mit der Methode nun einen weißen Kreis erzeugen können:

```
ngAfterViewInit(){
  this.container.createComponent(CircleComponent);
  this.container.createComponent(CircleComponent);
  this.addCircle('white');
}
```

Listing 5.27 »dynamic-components-demo.component.ts«: Aufruf der »addCircle«-Methode zum Anhängen eines weißen Kreises

Abbildung 5.17 zeigt das Ergebnis der Implementierung.

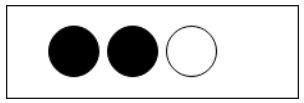

Abbildung 5.17 Kreis mit verändertem Input-Binding

Selbstverständlich ist es über die `ComponentRef` ebenfalls möglich, sich bei einem Output-Binding anzumelden oder sonstige Methoden der Komponente aufzurufen – Ihrer Kreativität sind hier keine Grenzen gesetzt.

5.3.3 Komponenten an einer bestimmten Stelle einfügen

Ohne weitere Parameter hängt die `createComponent`-Methode neu erzeugte Komponenten immer ans Ende an. Über einen weiteren Parameter haben Sie aber außerdem die Möglichkeit, zusätzliche Optionen an die `createComponent`-Methode zu übergeben. Die Eigenschaft `index` bestimmt hierbei den Index, an dem die Komponente eingefügt werden soll. Möchten Sie über die Methode `addCircle` erzeugte Komponenten beispielsweise immer vorne (an der ersten Position) einfügen, so können Sie dies wie folgt erreichen:

```
addCircle(color: string) {
  const circleRef = this.container.createComponent(CircleComponent,
                                                   {index: 0});
  circleRef.instance.color = color;
  return circleRef;
}
```

Listing 5.28 »dynamic-components-demo.component.ts«: Verwendung des »index«-Parameters

Der weiße Kreis wird nun, wie in Abbildung 5.18 zu sehen, vor den beiden schwarzen Kreisen im DOM-Baum positioniert.

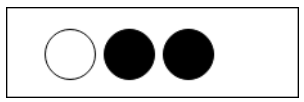

Abbildung 5.18 Hinzufügen der Komponente an erster Position

5.3.4 Komponenten innerhalb des ViewContainers verschieben und löschen

Des Weiteren ist es ebenfalls möglich, erzeugte Komponenten nachträglich im DOM-Baum zu verschieben bzw. sie wieder zu löschen. So können Sie sich über die `ViewCon-`

tainerRef-Methode get zunächst Zugriff auf die Komponente verschaffen, die sich an einer bestimmten Stelle im Container befindet. Über die Methode move können Sie die Komponente anschließend an einen beliebigen Index verschieben. Beide Methoden arbeiten dabei mit Instanzen der Klasse ViewRef:

```
moveCircle(oldIndex: number, newIndex: number) {
  const viewRef = this.container.get(oldIndex);
  this.container.move(viewRef, newIndex)
}
```

Listing 5.29 »dynamic-components-demo.component.ts«: Implementierung der »moveCircle«-Methode

Möchten Sie nun den ersten schwarzen Kreis an die erste Position verschieben, gehen Sie wie folgt vor:

```
ngAfterViewInit(){
  this.container.createComponent(CircleComponent);
  this.container.createComponent(CircleComponent);
  this.addCircle('white');
  this.moveCircle(1, 0);
}
```

Der weiße Kreis befindet sich nun, wie Abbildung 5.19 zeigt, in der Mitte.

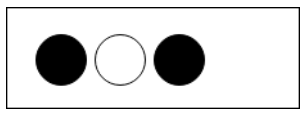

Abbildung 5.19 Darstellung nach dem Verschieben des schwarzen Kreises

Des Weiteren haben Sie außerdem die Möglichkeit, direkt über die Komponenten-Referenz Zugriff auf die ViewRef zu erhalten. Die Klasse ComponentRef stellt Ihnen hierfür die Eigenschaft hostView zur Verfügung:

```
ngAfterViewInit(){
  ...
  const circleRef = this.addCircle('gray');
  this.container.move(circleRef.hostView, 1);
}
```

Listing 5.30 »dynamic-components-demo.component.ts«: Verwendung der »hostView«-Eigenschaft der »Component«-Referenz

Wie erwartet wird der graue Kreis nun an zweiter Stelle dargestellt, wie Sie in Abbildung 5.20 sehen.

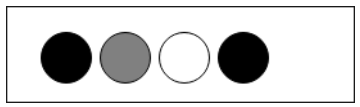

Abbildung 5.20 Darstellung nach dem Verschieben des grauen Kreises

Zu guter Letzt bietet Ihnen die Klasse ViewContainerRef mit der remove-Methode die Möglichkeit, Komponenten über ihren Index aus dem DOM-Baum zu entfernen. Möchten Sie beispielsweise den letzten Kreis entfernen, so können Sie dies mit der remove-Methode sowie mit der length-Eigenschaft wie folgt erreichen:

```
this.container.remove(this.container.length - 1);
```

Hierdurch wird die Komponente einerseits aus der View entfernt und andererseits ordnungsgemäß beendet. So haben Sie über den ngOnDestroy-Callback auch für dynamische Komponenten die Möglichkeit, notwendige Aufräumarbeiten durchzuführen.

Komponenten mit der »detach«-Methode aushängen

Bei der Recherche im Netz sind Sie unter Umständen über das ein oder andere Online-Tutorial gestolpert, in dem dynamisch erzeugte Komponenten mithilfe der ViewContainerRef-Methode detach entfernt werden.

Beachten Sie hierbei, dass detach die Komponente zwar aus dem DOM-Baum entfernt, diese aber weiterhin »am Leben« ist! So können Sie per detach entfernte Komponenten zu einem späteren Zeitpunkt über die insert-Methode wieder in den Baum einhängen. Möchten Sie eine Komponente wirklich entfernen, sollten Sie also auf jeden Fall die remove-Methode einsetzen!

5.3.5 createEmbeddedView: eigene strukturelle Direktiven implementieren

Neben dem Hinzufügen von Komponenten-Instanzen bietet Ihnen die Klasse ViewContainerRef über die createEmbeddedView-Methode außerdem die Möglichkeit, Template-Referenzen zur Laufzeit zu Ihrer View hinzuzufügen. So können Sie über diese Technik leicht eigene *strukturelle Direktiven* erstellen, also Direktiven, die zur Laufzeit neue DOM-Knoten erzeugen.

Die Repeater-Direktive

Möchten Sie beispielsweise eine Repeater-Direktive implementieren, die ein Template mehrfach darstellt, so tun Sie dies mithilfe von createEmbeddedView wie folgt:

```
@Directive({
  selector: '[chRepeater]'
})
```

```
export class RepeaterDirective {
  constructor(private container: ViewContainerRef,
              private template: TemplateRef<any>) {
  }
  @Input('chRepeater') set repeatIt(count: number) {
    this.container.clear();
    for (let i = 0; i < count; i++) {
      this.container.createEmbeddedView(this.template);
    }
  }
}
```

Listing 5.31 »repeater.directive.ts«: Implementierung der strukturellen Repeater-Direktive

Die Direktive erhält die Referenzen auf die `ViewContainerRef` sowie die `TemplateRef` direkt über den Konstruktor (und nicht wie in den vorigen Beispielen über `ContentChildren`-Querys). Die Direktive erwartet durch diese Implementierung, dass sie direkt auf einem `<ng-template>`-Element eingesetzt wird. Das Input-Binding wird an dieser Stelle über eine ES2015-Setter-Methode (siehe Anhang A, »ECMAScript 2015 (and beyond)«) implementiert, sodass die Logik der `repeatIt`-Methode jedes Mal ausgeführt wird, wenn sich das Binding ändert.

Innerhalb der Methode wird der Container zunächst geleert. Anschließend wird das Template so oft wie gewünscht über die `createEmbeddedView`-Methode zur View hinzugefügt. Im HTML-Code können Sie die Direktive nun wie folgt verwenden:

```
<input type="number" [(ngModel)]="repeatCnt"> <br><br>
<ng-template [chRepeater]="repeatCnt">
  <div>
    <i>Ich werde wiederholt!</i>
  </div>
</ng-template>
```

Listing 5.32 »dynamic-components-demo.component.ts«: Verwendung der Repeater-Direktive in der Anwendung

Das `div`-Element, das Sie innerhalb des `<ng-template>`-Tags definiert haben, wird nun so oft wiederholt, wie der Wert im Input-Feld angibt (siehe Abbildung 5.21).

Abbildung 5.21 Die Repeater-Direktive im Einsatz

Doch Moment: Eventuell erinnert der Aufbau von Listing 5.32 Sie bereits an die Beschreibungen zur Templating-Microsyntax aus Kapitel 3, »Komponenten und Templating: der Angular-Sprachkern«. Dort habe ich erklärt, dass die *-Syntax lediglich eine Kurzschreibweise für den Einsatz von HTML5-Templates ist und dass die beiden Schreibweisen

```
<div *ngIf="isVisible" > Hallo NgIf </div>
```

und

```
<ng-template [ngIf]="isVisible">
 <div> Hallo NgIf </div>
</ng-template>
```

funktional vollkommen identisch sind. So haben Sie – ohne es gemerkt zu haben – soeben Ihre erste strukturelle, Microsyntax-kompatible Direktive entwickelt. Ändern Sie Listing 5.32 somit wie folgt, und freuen Sie sich über einen weiteren Baustein in Ihrer Angular-Sprachkiste:

```
<div *chRepeater="repeatCnt">
  <i>Ich werde wiederholt!</i>
</div>
```

Der Template-Kontext: Übergabe von Parametern an dynamisch erzeugte Templates

In manchen Fällen kann es des Weiteren notwendig sein, Parameter an ein dynamisch erzeugtes Template zu übergeben. Da reine Template-Elemente allerdings keine Input-Bindings oder Ähnliches unterstützen, müssen Sie hier eine etwas eigenwillige Schreibweise verwenden. So haben Sie im vorigen Abschnitt gesehen, wie Sie Eingangsparameter von Templates über die `let`-Schreibweise definieren können:

```
<ng-template let-entry>
  ...
  <img [src]="entry?.image" [alt]="entry?.title" class="small"/>
</ng-template>
```

Listing 5.33 »blog.component.html«: Übergabe der »entry«-Variablen im Blog-Beispiel

Im Zusammenspiel mit der `createEmbeddedView`-Methode benötigen Sie nun aber noch eine Möglichkeit, diesen Eingangsparameter dynamisch zu beschreiben. Dazu können Sie in der Methode über einen zweiten Parameter den sogenannten *Template-Kontext* definieren. Möchten Sie beispielsweise ein dynamisch erzeugtes Todo-Template befüllen, so können Sie dies wie folgt implementieren:

```
export class DynamicComponentsDemoComponent {
  @ViewChild('todoContainer', {read: ViewContainerRef}) todoContainer!:
ViewContainerRef;
```

```
  @ViewChild('todoTemplate') todoTemplate!: TemplateRef<any>;
  ...
  ngAfterContentInit() {
    this.todoContainer.createEmbeddedView(this.todoTemplate, {
      todoParam: {
        text: 'Aufräumen',
        done: true
      }
    })
  }
}
```

Listing 5.34 »dynamic-components-demo.component.ts«:
Übergabe eines Kontext-Objekts an die »createEmbeddedView«-Methode

So enthält der Template-Kontext nun ein Objekt mit dem Schlüssel `todoParam`. Innerhalb des eigentlichen Templates müssen Sie dieses Objekt nun nur noch dem `let-todo`-Eingangsparameter zuweisen:

```
<ng-template #todoTemplate let-todo="todoParam">
 <div [ngStyle]="{'text-decoration': todo.done ? 'line-through' : ''}">
   <b>{{todo.text}}</b>
 </div>
</ng-template>
```

Listing 5.35 »dynamic-components-demo.component.html«:
Zuweisung der Template-Kontextvariablen

Abbildung 5.22 zeigt, dass das dynamisch geladene Template nun mit den Daten aus dem Kontext-Parameter befüllt wird.

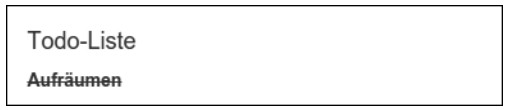

Abbildung 5.22 Darstellung des »todoTemplate«
nach der Befüllung über den Template-Kontext

Missbrauch von »createEmbeddedView« für Applikationslogik

Beachten Sie dabei, dass Sie den obigen Anwendungsfall in einer echten Anwendung deutlich eleganter mit Standardsprachmitteln wie `ngFor` in Verbindung mit einer Membervariablen implementieren würden.

Um den Templating-Mechanismus wirklich zu verstehen, ist es aber dennoch sinnvoll, die Konzepte zu kennen, die dem View-Container zugrunde liegen.

5.4 NgComponentOutlet: dynamisch erzeugte Komponenten noch einfacher verwalten

Die Möglichkeit, Komponenten mithilfe von `ViewContainerRef` zur Laufzeit zum DOM-Baum hinzuzufügen, ist sehr flexibel. Möchten Sie aber lediglich eine einzelne dynamisch erzeugte Komponente an einer bestimmten Stelle im DOM-Baum einfügen, bietet Angular Ihnen mit der `NgComponentOutlet`-Direktive eine noch etwas komfortablere Technik an. Listing 5.36 und Listing 5.37 zeigen exemplarisch das dynamische Einfügen der `CircleComponent` mithilfe der Direktive:

```
export class DynamicComponentsDemoComponent {
  circleComponent = CircleComponent;
  ...
}
```

Listing 5.36 »dynamic-components-demo.component.ts«:
Speichern der »CircleComponent«-Klasse in einer Membervariablen

```
<div *ngComponentOutlet="circleComponent"></div>
```

Listing 5.37 »dynamic-components-demo.component.html«:
Rendern der »CircleComponent« mithilfe von »NgComponentOutlet«

Wie Sie sehen, müssen Sie die Komponenten-Klasse, die Sie rendern möchten, zunächst in einer Membervariablen speichern. Dies ist notwendig, da Sie ansonsten nicht aus dem Template heraus auf die Klasse zugreifen können. Listing 5.37 zeigt anschließend den Einsatz der `NgComponentOutlet`-Direktive, um die Komponente an einer bestimmten Stelle zu rendern.

Gegebenenfalls wundern Sie sich an dieser Stelle, wieso die Komponente hier nicht einfach direkt über Ihr Tag `ch-circle` in die Anwendung eingefügt wurde. Der Vorteil von `NgComponentOutlet` wird dann deutlich, wenn Sie die darzustellende Komponente dynamisch zur Laufzeit ändern möchten. Möchten Sie beispielsweise abhängig von einer Bedingung entweder einen Kreis oder ein Quadrat zeichnen, so ist dies mit `NgComponentOutlet` ein Kinderspiel:

```
export class DynamicComponentsDemoComponent {
  geoComponent: any = CircleComponent;
  toggleGeoComponent() {
    this.geoComponent = this.geoComponent === CircleComponent
                      ? SquareComponent : CircleComponent;
  }
}
```

Listing 5.38 »dynamic-components-demo.component.ts«:
Dynamisches Ändern der darzustellenden Geo-Komponente

```
<div *ngComponentOutlet="geoComponent"></div>
<button class="btn" (click)="toggleGeoComponent()">
  Geo-Komponente umschalten
</button>
```

Listing 5.39 »dynamic-components-demo.component.html«:
Darstellung der Geo-Komponente im Template

Über die Methode `toggleComponent` weisen Sie der Variablen `geoComponent` entweder die Komponente `CircleComponent` oder die Komponente `SquareComponent` zu. Ein Klick auf den Button GEO-KOMPONENTE UMSCHALTEN fügt nun abwechselnd entweder einen Kreis oder ein Quadrat zum DOM-Baum hinzu (siehe Abbildung 5.23).

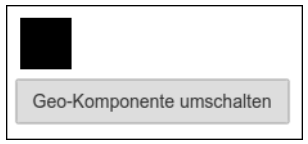

Abbildung 5.23 Darstellung des Quadrats nach Klick auf den Button

5.4.1 Übergabe von dynamischen Eigenschaften an NgComponentOutlet

Ein kleiner Nachteil bei der Verwendung von `NgComponentOutlet` besteht darin, dass es nicht möglich ist, auf die Membervariablen bzw. auf die Input-Bindings der darzustellenden Komponente zuzugreifen. Möchten Sie eine per `NgComponentOutlet` gerenderte Komponente dynamisch konfigurieren, müssen Sie an dieser Stelle auf eine Technik zurückgreifen, die Sie eigentlich erst in Kapitel 7, »Services und Dependency-Injection: lose Kopplung für Ihre Business-Logik«, kennenlernen werden.

So bietet Angular Ihnen mit dem Dependency-Injection-Framework eine Möglichkeit, Daten über den Konstruktor in eine Klasse zu injizieren. Diese Injektion erfolgt vereinfacht gesagt über sogenannte *Provider*, die bei einem *Injector* registriert werden. `NgComponentOutlet` erlaubt es an dieser Stelle, einen eigenen Injector an die Direktive zu übergeben, der die benötigten Abhängigkeiten und Daten für die gerenderte Komponente enthält. Listing 5.40 zeigt zunächst einmal die Implementierung der Komponente `DynamicDialogComponent` sowie der Konfigurationsklasse `DialogConfig`:

```
export class DialogConfig {
  title = '';
  text = '';
  callbackFunction?: (data: any) => void;
}

@Component({
  template: `
```



```
    <div class="panel panel-default">
      <div class="panel-heading">{{config.title}}</div>
      <div class="panel-body">{{config.text}}</div>
      <div class="panel-footer">
        <button class="btn btn-sm" (click)="confirm(true)">OK</button>
        <button class="btn btn-sm" (click)="confirm(false)">Abbrechen</button>
      </div>
    </div>
  `,
})
export class DynamicDialogComponent  {
  constructor(public config: DialogConfig) { }
  confirm(result: boolean) {
    this.config.callbackFunction?.(result);
  }
}
```

Listing 5.40 »dynamic-components-demo.component.ts«:
Implementierung einer dynamischen Dialog-Komponente

Wie Sie sehen, enthält die Klasse `DialogConfig` die beiden Konfigurationseigenschaften `title` und `text` sowie eine optionale `callbackFunction`. Die Klasse soll dazu verwendet werden, die Komponente `DynamicDialogComponent` zu konfigurieren. Dies erfolgt dadurch, dass die Konfiguration über den Konstruktor in die Komponente hineingegeben wird:

```
constructor(public config: DialogConfig) { }
```

Aufruf von optionalen Funktionen

Beachten Sie in Listing 5.40 auch den Aufruf der optionalen Callback-Funktion:

`this.config.callbackFunction?.(result);`

So erlaubt TypeScript die Verwendung der *Optional-Chaining-Syntax* auch beim Aufruf von optionalen Funktionen!

Das Rendern dieser Komponente mithilfe von `NgComponentOutlet` benötigt nun eine zugegebenermaßen auf den ersten Blick etwas eigenwillige Syntax. Dabei ist es an dieser Stelle aber nicht notwendig, dass Sie jede technische Feinheit verstehen – Details zum Dependency-Injection-Framework werden Sie, wie bereits beschrieben, in Kapitel 7 lernen! Der erste Schritt besteht hier darin, einen eigenen Injector zu definieren, der eine Instanz der Klasse `DialogConfig` bereitstellt. Listing 5.41 zeigt die entsprechende Implementierung in der Klasse `DynamicComponentsDemoComponent`:

```
export class DynamicComponentsDemoComponent {
  ...
  dialogInjector: Injector;
  dialogComponent = DynamicDialogComponent;
  constructor(private injector: Injector) {
    const dialogConfig: DialogConfig = {
      title: 'Eintrag löschen',
      text: 'Wollen Sie den Eintrag wirklich löschen?',
      callbackFunction: (result) => {
        if (result) {
          console.log('Eintrag wurde gelöscht');
        }
      }
    };
    this.dialogInjector = Injector.create({
      providers: [
        {provide: DialogConfig, useValue: dialogConfig}
      ],
      parent: injector
    });
  }
}
```

Listing 5.41 »dynamic-components-demo.component.ts«:
einen eigenen Injector für den dynamischen Dialog erzeugen

Über den Ausdruck

```
const dialogConfig: DialogConfig = {
  title: 'Eintrag löschen',
  text: 'Wollen Sie den Eintrag wirklich löschen?',
  ...
};
```

wird hier zunächst eine Instanz der `DialogConfig`-Klasse erzeugt.

Die Funktion `Injector.create` erzeugt anschließend einen neuen Injector, der eine Instanz der Klasse `DialogConfig` bereitstellt (*provided*) und hierfür den Wert der zuvor angelegten Variablen benutzt:

```
this.dialogInjector = Injector.create({
  providers: [
    {provide: DialogConfig, useValue: dialogConfig}
  ],
  parent: injector
});
```


Beachten Sie dabei, dass Sie den Injector der Demo-Komponente über die `parent`-Eigenschaft an den neuen Injector übergeben. Dies ist notwendig, um den neuen Injector korrekt in den Injector-Baum einzuhängen.

Um den neu erzeugten Injector zu verwenden, müssen Sie nun nichts weiter tun, als diesen bei der Verwendung der `NgComponentOutlet`-Direktive zu referenzieren:

```
<div *ngComponentOutlet="dialogComponent;
                         injector: dialogInjector;">
</div>
```

Listing 5.42 »dynamic-components-demo.component.html«:
Übergabe des neuen Injectors an die »NgComponentOutlet«-Direktive

Ein Blick in die Oberfläche zeigt, dass der dargestellte Dialog nun mit den übergebenen Werten konfiguriert wurde. Ein Klick auf den OK-Button gibt wie erwartet den Text »Eintrag wurde gelöscht« in der Developer-Konsole aus (siehe Abbildung 5.24).

Abbildung 5.24 Darstellung des dynamisch erzeugten Dialogs

5.5 ChangeDetection-Strategien: Performance-Boost für Ihre Applikation

Der Angular-ChangeDetection-Mechanismus bestimmt, wann bzw. wie oft Ihre Komponenten auf Änderungen untersucht werden: Ändern sich die Daten der Applikation, so muss das Framework betroffene Komponenten untersuchen und gegebenenfalls neu zeichnen. Wenn Sie – besonders bei der Definition Ihrer Data-Bindings und der Konfiguration der ChangeDetection – einige einfache Regeln einhalten, können Sie hier beachtliche Performance-Steigerungen erzielen.

Kapitel 12
Reaktive Architekturen mit RxJS

»Everything is a stream« – getreu dem Motto der reaktiven Entwicklergemeinde werden Sie in diesem Kapitel neue Denkansätze zur Entwicklung und Architektur Ihrer Anwendung erhalten.

Nachdem Sie in den vorangegangenen Kapiteln schon in verschiedenen Bereichen Kontakte mit der Bibliothek *RxJS* und der `Observable`-Klasse geknüpft haben, geht es in diesem Kapitel reaktiv ans Eingemachte. So haben Sie beispielsweise in Kapitel 11, »HTTP: Anbindung von Angular-Applikationen an einen Webserver«, Techniken kennengelernt, mit denen Sie asynchron geladene Daten sehr elegant auf Ihrer Oberfläche darstellen können (Stichwort `AsyncPipe`).

Alle bisherigen Ansätze basierten aber immer noch auf einem klassischen (Pull-basierten) Programmiermodell: Bei der Eingabe in ein Suchfeld wurde die `findTasks`-Methode aufgerufen. Anschließend wurde das Ergebnis dieses Aufrufs an eine Klassenvariable gebunden und dadurch in der Oberfläche ausgegeben.

Komponenten, die Daten benötigen, »ziehen« sich diese also dann, wenn sie sie benötigen. Grundsätzlich ist dies vollkommen in Ordnung. Der Einsatz von reaktiven Technologien bietet Ihnen jedoch Möglichkeiten, Ihre Anwendung sowohl deutlich flexibler als auch performanter zu gestalten. So werden Sie in diesem Kapitel unter anderem lernen,

- welche Rolle die grundlegenden Bestandteile *Observables* und *Observer* in einer reaktiven Anwendung spielen.
- wie Sie mit wenigen Zeilen Code eine *Typeahead*-Suche implementieren und dabei die reaktiven Bestandteile, die von den `Forms`-, `Routing`- und `HttpClient`-Modulen bereitgestellt werden, elegant miteinander verbinden können.
- was eine »Push-getriebene« Datenarchitektur ist und welche Vorteile Ihnen diese bei der Entwicklung einer komplexen Anwendung bieten kann.
- welche grundsätzlichen Ideen hinter dem Schlagwort *Redux* stehen und wie Sie diese Technik elegant mit Observable-Streams kombinieren können.
- wie Sie Ihre Anwendung mithilfe von *Websockets* ohne großen Aufwand echtzeitfähig machen können.

Die Arbeit mit einem reaktiven Programmiermodell erfordert anfangs ein gewisses Maß an Umdenken. Ich verspreche Ihnen jedoch, dass sich die investierte Energie lohnen wird: Die Integration von reaktiven Technologien in die neue Angular-Plattform ist definitiv einer der innovativsten Bestandteile des Frameworks!

Hinweis zu den Beispielquelltexten

Sie finden die fertigen Beispielquelltexte dieses Kapitels im Verzeichnis *project-manager-reactive*. Möchten Sie die einzelnen Schritte selbst nachimplementieren, so empfehle ich Ihnen, das in Kapitel 11 vorgestellte Beispiel (*project-manager-http*) als Basis zu verwenden. Wie in allen Beispielen erfolgt der Start der Applikation mithilfe der Befehle `npm install` und `npm start`.

12.1 Kurzeinführung in RxJS

Allein über die Bibliothek *RxJS* könnte man ganze Bücher schreiben. So stellt RxJS Ihnen eine schier endlose Anzahl an vordefinierten Observable-Typen und Operatoren zur Realisierung von reaktiven Anwendungen zur Verfügung. Sie alle im Detail vorzustellen, würde den Rahmen dieses Kapitels sprengen.

Ich habe mich daher entschieden, Ihnen in diesem ersten Abschnitt lediglich eine kurze Übersicht über die wichtigsten Konzepte der Bibliothek zu geben. Möchten Sie sich im Anschluss selbst noch intensiver mit RxJS beschäftigen, steht Ihnen auf der Homepage des Projekts

https://rxjs.dev/

eine wirklich gute Online-Dokumentation zur Verfügung.

12.1.1 Observables und Observer-Functions: die Kernelemente der reaktiven Programmierung

Für die professionelle Arbeit mit RxJS ist es nützlich, sich zunächst einmal mit den Kernelementen der Bibliothek vertraut zu machen. So haben Sie in den bisherigen Kapiteln immer mit Observables gearbeitet, die Angular bereitstellt. In diesem Abschnitt geht es nun darum, eigene Observables zu erzeugen und die absolute Basisfunktionalität von RxJS am Beispiel der beiden Klassen `Observable` und `Observer` kennenzulernen.

Der direkteste Weg, ein *Observable* zu erzeugen, besteht in der Verwendung des Schlüsselworts `new`. Der Konstruktor nimmt als Parameter eine Funktion entgegen, mit deren Hilfe Sie Daten an den sogenannten `Observer` – also den Datenkonsumenten – senden können. Listing 12.1 zeigt die Verwendung sowie die Anmeldung beim erzeugten Observable über die `subscribe`-Methode:


```
const observable = new Observable((observer: Observer<number>) => {
  observer.next(1);
  observer.next(2);
  observer.next(3);
  observer.complete()
});
observable.subscribe(
  (value) => { console.log('new value: ', value);  },
  (error) => { console.log('error: ', error);  },
  () => { console.log('completed successfully');
});
```

Listing 12.1 »rxdemo.component.ts«: Low-Level-Erzeugung eines Observables und Verarbeitung der Daten des Streams

Während der Lebensdauer eines Observables können im Wesentlichen drei Dinge passieren:

- Das Observable stellt einen neuen Wert bereit.
- Innerhalb des Observables tritt ein Fehler auf.
- Das Observable wird beendet.

Um neue Werte auf den Datenstrom (Stream) zu schicken, Fehler auszulösen oder den Stream zu beenden, stehen Ihnen dabei die drei Methoden `next`, `error` und `complete` zur Verfügung. Mit dem Ausdruck

```
observer.next(1);
```

sagen Sie dem Framework also im Endeffekt nichts anderes als: »Schicke den Wert `1` auf den Stream.«

Die Reaktion auf neue Werte im `Observable` geschieht folgerichtig durch die Registrierung von (maximal) drei Funktionen über die `subscribe`-Methode. So führt Listing 12.1 zunächst einmal zur folgenden Konsolenausgabe:

```
new value: 1
new value: 2
new value: 3
completed successfully
```

Die Registrierung der Funktionen an der `subscribe`-Methode wird Ihnen an dieser Stelle bereits bekannt vorkommen: Sie haben diesen Mechanismus beispielsweise bereits bei der Verwendung des `HttpClient`-Moduls zur Unterscheidung von erfolgreichen Requests und nicht erfolgreichen Requests verwendet. Hätten Sie bei Ihrer Implementierung noch die dritte `complete`-Funktion hinzugefügt, hätten Sie dort

außerdem gesehen, dass die von der *Http*-Bibliothek bereitgestellten Observables direkt nach der Verarbeitung der Response beendet werden. Es handelt sich somit um sogenannte *Single-value Observables*, die nach dem Absenden eines Werts sofort beendet werden.

12.1.2 Subscriptions, der takeUntil-Operator und Disposing-Functions: Observables sauber beenden

Im vorigen Beispiel wurde das Observable dadurch beendet, dass der Datenproduzent die `complete`-Funktion aufgerufen hat. Gerade bei »unendlich« lange laufenden Observables sollten Sie allerdings dafür sorgen, dass Sie sich manuell vom `Observable` abmelden, wenn Sie nicht mehr an dessen Daten interessiert sind. Die direkteste Möglichkeit, dies zu tun, besteht in der Verwendung des `Subscription`-Objekts, das die `subscribe`-Methode zurückliefert. Eine Abmeldung erfolgt dabei durch den expliziten Aufruf der `unsubscribe`-Funktion auf dieser `Subscription`.

Listing 12.2 zeigt die Implementierung eines `Observable`, das jede Sekunde die aktuelle Uhrzeit in der Variablen `currentDate` speichert. Wird die Seite verlassen (`ngOnDestroy`), wird die `Subscription` beendet.

```
import {Observable, Subscription, timer} from 'rxjs';
...
export class RxDemoComponent {
  dateSubscription: Subscription;
  currentDate: Date;
  ngOnInit() {
    this.dateSubscription = timer(0, 1000)
      .pipe(map(() => new Date()))
      .subscribe(value => {
        this.currentDate = value;
      });
  }
  ngOnDestroy() {
    this.dateSubscription.unsubscribe();
  }}
```

Listing 12.2 »rxdemo.component.ts«: Abmeldung vom Observer beim Verlasssen der Seite

Erzeugungsoperatoren in RxJS

In Listing 12.2 sehen Sie mit der `timer`-Funktion einen von RxJS bereitgestellten Erzeugungsoperator (*Creation Operator*) in Aktion. Die `timer`-Funktion erzeugt ein Observable, das zum ersten Mal nach Ablauf des initialen Timeouts (im Beispiel `0` Millisekunden) und dann jeweils nach Ablauf des vorgegebenen Intervalls (`1000` Millisekunden) einen neuen Wert auf den Stream schickt.

Die erzeugten Observables lassen sich – wie im Listing zu sehen – anschließend mithilfe der Ihnen schon bekannten `pipe`-Funktion in Verbindung mit Operatoren (wie dem `map`-Operator) manipulieren.

Neben der `timer`-Funktion stellt RxJS Ihnen ca. 20 weitere Operatoren zur Erzeugung von Observables für die verschiedensten Anwendungsfälle zur Verfügung. Sollten Sie sich näher mit den hier möglichen Optionen auseinandersetzen wollen, empfehle ich Ihnen erneut einen Blick in die Online-Dokumentation der Bibliothek:

https://rxjs-dev.firebaseapp.com/api

Der takeUntil-Operator: elegantes Subscription-Handling leicht gemacht

Insbesondere dann, wenn Ihre Komponente mehrere Observable-Streams verwendet, kann das manuelle Verwalten der `Subscription`-Objekte schnell aufwendig und fehleranfällig werden.

Eine Alternative besteht in diesem Fall in der Verwendung des `takeUntil`-Operators. Der Operator nimmt als einzigen Parameter ein weiteres Observable entgegen. Sobald dieses innere Observable einen Wert emittet, wird das »äußere« Observable beendet. Schauen Sie sich für ein besseres Verständnis des Operators den Code aus Listing 12.3 an.

```
export class RxDemoComponent implements OnInit, OnDestroy {
  destroyed$ = new Subject<void>();
  ...
  ngOnInit() {
    timer(0, 1000).pipe(
      map(() => new Date()),
      takeUntil(this.destroyed$)
    )
    .subscribe(value => {
      this.currentDate = value;
    });
    ...
  }

  ngOnDestroy() {
    this.destroyed$.next();
    ...
  }
  ...
}
```

Listing 12.3 »rxdemo.component.ts«: Verwendung des »takeUntil«-Operators

Über die Zeile

```
destroyed$ = new Subject<void>();
```

instanziieren Sie an dieser Stelle das `Subject` `destroyed$`. Ich stelle Ihnen Subjects im Detail im folgenden Abschnitt vor. Für den Moment reicht es aber, zu wissen, dass Subjects eine spezielle Form von Observables sind, die Ihnen zusätzlich die Möglichkeit bieten, über die `next`-Methode neue Werte auf den Stream zu schicken. So sorgt der Code

```
ngOnDestroy() {
  this.destroyed$.next();
  ...
}
```

dafür, dass der `destroyed$`-Stream bei der Zerstörung der Komponente einen Wert emittet.

Bei der Definition eines neuen Observables können Sie nun den `destroyed$`-Stream in Verbindung mit dem `takeUntil`-Operator verwenden, um sicherzustellen, dass das `Observable` bei der Zerstörung der Komponente korrekt beendet wird:

```
timer(0, 1000).pipe(
  map(() => new Date()),
  takeUntil(this.destroyed$)
)
```

Subscriptions und die AsyncPipe

An dieser Stelle ist es außerdem interessant zu wissen, dass Sie sich bei der Verwendung der `AsyncPipe` keine Gedanken über die manuelle Abmeldung von einem `Observable` machen müssen.

Listing 12.4 zeigt die Definitionen eines Streams, der immer die aktuelle Uhrzeit enthält. Beachten Sie dabei insbesondere, dass innerhalb des Komponenten-Codes keine Anmeldung (`subscribe`) am `Observable` stattfindet:

```
currentTime$: Observable<Date>;
ngOnInit() {
  this.currentTime$ = timer(0, 1000).pipe(map(() => new Date()));
}
```

Listing 12.4 »rxdemo.component.ts«: Stream, der immer die aktuelle Uhrzeit enthält

Möchten Sie die Uhrzeit nun, wie in Abbildung 12.1 dargestellt, auf der Oberfläche anzeigen, kann dies sehr elegant mithilfe der `AsyncPipe` erfolgen:


```
Aktuelle Uhrzeit: {{currentTime$ | async | date:"HH:mm:ss"}}
```

Listing 12.5 »rxdemo.component.html«: Auslesen und Formatieren der aktuellen Uhrzeit in der Oberfläche

Abbildung 12.1 Darstellung der aktuellen Uhrzeit in der Oberfläche

In diesem Fall meldet sich die `AsyncPipe`, sobald die Seite dargestellt wird, beim `currentTime$`-Stream. Beim Verlassen der Seite und bei der damit verbundenen Zerstörung der Komponente wird anschließend automatisch die `unsubscribe`-Methode aufgerufen.

Low-Level-Observables und Angular

Listing 12.4 und Listing 12.5 zeigen sehr schön, dass sich die nahtlose Integration von Observables in Angular nicht nur auf die Verwendung von HTTP-Aufrufen oder Formularwerten beschränkt. Mechanismen wie die `AsyncPipe` funktionieren auch problemlos mit »Low-Level«-Observables.

Disposing Functions: eigene Observables sauber beenden

Möchten Sie in Ihren selbst implementierten Observables auf die Abmeldung von diesem `Observable` reagieren, so können Sie dies durch die Rückgabe einer sogenannten *Disposing Function* im Konstruktor tun. Die Funktion bietet Ihnen die Möglichkeit, durch das `Observable` allozierte Ressourcen wieder freizugeben oder zu beenden.

Listing 12.6 zeigt exemplarisch die Implementierung eines `Observable`, das jede Sekunde einen zufälligen Wert auf den Stream schickt. Die Umsetzung erfolgt dabei auf Basis der globalen `setInterval`-Funktion, die es erforderlich macht, das erzeugte Intervall mithilfe der `clearInterval`-Funktion zu beenden:

```
randomValues$ = new Observable((observer) => {
  const interval = setInterval(() => {
    observer.next(Math.random());
  }, 1000);
  return () => {
    clearInterval(interval);
  }});
```

Listing 12.6 »rxdemo.component.ts«: Bereitstellung einer Disposing Function als Rückgabewert bei der Observable-Erzeugung

Sie können das `randomValues$`-Observable nun wie gewohnt dazu verwenden, zufällige Werte in der Oberfläche darzustellen. Entscheiden Sie sich zu einem späteren Zeitpunkt, die entsprechende `Subscription` über die `unsubscribe`-Funktion oder den `takeUntil`-Operator zu beenden, sorgt RxJS automatisch dafür, dass die Disposing Function aufgerufen und das Intervall sauber beendet wird:

```
startRandomValuesObservable() {
  this.randomValuesSub = this.randomValues$.subscribe((value) => {
    this.randomValue = value;
  });
}
stopRandomValuesObservable() {
  this.randomValuesSub.unsubscribe(); // clearing interval
}
```

Listing 12.7 »rxdemo.component.ts«: Verwendung des »randomValues«-Observables

12.1.3 Subjects: Multicast-Funktionalität auf Basis von RxJS

Bei *Subjects* handelt es sich um eine spezielle Form von Observables, die es Ihnen ermöglicht, auf sehr elegante Art und Weise *Publish-Subscribe*-Funktionalität zu implementieren. Subjects unterscheiden sich dabei im Wesentlichen durch zwei grundlegende Punkte von einfachen Observables:

1. Ein `Subject` kann mehrere `Subscriber` besitzen:

   Während der Aufruf der `subscribe`-Funktion an einem regulären `Observable` immer dazu führt, dass der Subscriber einen eigenen Ausführungskontext erhält, teilen sich bei einem `Subject` alle `Subscriber` die gleiche Ausführung.

2. Ein `Subject` ist immer gleichzeitig auch ein Datenproduzent:

   Im Gegensatz zu anderen Observables besitzen Subjects selbst die Methoden `next`, `error` und `complete`. Sie sind somit gleichzeitig `Observable` und `Observer`.

Schauen Sie sich, um die Unterschiede besser zu verstehen, zunächst noch einmal die folgende Verwendung des `randomValues$`-Observables aus Abschnitt 12.1.2 an:

```
this.randomValues$ = new Observable((observer: Observer<number>) => {
  // siehe voriger Abschnitt
});
this.sub1 = this.randomValues$.subscribe((value) => {
  console.log(`Subscription 1: ${value}`);
});

this.sub2 = this.randomValues$.subscribe((value) => {
```



```
  console.log(`Subscription 2: ${value}`);
});
```

Listing 12.8 »rxdemo.component.ts«: Anmeldung von zwei Observern beim »randomValues«-Observable

Ein Blick in die Developer-Konsole zeigt, dass in diesem Beispiel jeder `Observer` eigene zufällige Werte – und somit einen eigenen Ausführungskontext – erhält:

```
Subscription 1: 0.9913120189674176
Subscription 2: 0.6696771499303198
Subscription 1: 0.08140152984303839
Subscription 2: 0.8133887466413385
Subscription 1: 0.5074920971798378
Subscription 2: 0.04292412726971806
```

Die Unterschiede zwischen Subjects und einfachen Observables werden deutlich, wenn man sich die äquivalente Umsetzung der Logik auf Basis eines `Subject` anschaut:

```
this.randomValuesSubject$ = new Subject<number>();
const interval = setInterval(() => {
  this.randomValuesSubject$.next(Math.random());
}, 1000);

this.sub1 = this.randomValuesSubject$.subscribe((value) => {
  console.log(`Subscription 1: ${value}`);
});

this.sub2 = this.randomValuesSubject$.subscribe((value) => {
  console.log(`Subscription 2: ${value}`);
});
```

Listing 12.9 »rxdemo.component.ts«: Definition eines Subjects mit zwei Subscribern und das Erzeugen von zufälligen Werten für dieses Subject

Wie Sie sehen, erfolgt die Erzeugung der Daten hier durch den direkten Aufruf der `next`-Funktion des `Subject`. Schauen Sie nun in die Developer-Konsole, werden Sie feststellen, dass beide `Subscriber` die gleichen Werte erhalten. Sie teilen sich somit den Ausführungskontext des `Subject`:

```
Subscription 1: 0.18943417481883795
Subscription 2: 0.18943417481883795
Subscription 1: 0.7449478603959496
Subscription 2: 0.7449478603959496
```

```
Subscription 1: 0.6799151167214585
Subscription 2: 0.6799151167214585
```

Subjects bieten Ihnen somit eine komfortable Möglichkeit, Daten an mehrere interessierte `Subscriber` zu verteilen.

Vordefinierte Subject-Arten in RxJS

Neben der Klasse `Subject` stellt RxJS Ihnen außerdem einige weitere vordefinierte Subject-Arten zur Verfügung. So werden Sie beispielsweise bei der Umsetzung der alternativen Datenarchitektur in Abschnitt 12.3 auf die Klasse `BehaviorSubject` zurückgreifen, die sich insbesondere dann anbietet, wenn während der Existenz des `Subject` neue Subscriber hinzukommen oder wegfallen.

12.2 Implementierung einer Typeahead-Suche

Nun aber genug der Theorie! In diesem Abschnitt möchte ich Ihnen ein klassisches Beispiel für die Verwendung von Observable-Streams vorstellen: die Implementierung einer Typeahead-Komponente. Neben der eigentlichen Typeahead-Funktionalität werden Sie in diesem Beispiel aber außerdem sehen, wie Sie die einzelnen Teilbereiche Routing, Formulare und HTTP äußerst elegant miteinander verbinden können. Schauen Sie sich als Einstieg zunächst noch einmal die bisherige Implementierung des Suchfeldes an:

```
<input type="text" #query class="form-control"
       (keyup.enter)="findTasks(query.value)"
       [formControl]="searchTerm">
```

Listing 12.10 »task-list.component.html«: bisherige Implementierung des Input-Suchfeldes

```
tasks$: Observable<Task[]>;
...
findTasks(queryString: string) {
  this.tasks$ = this.taskService.findTasks(queryString);
  this.adjustBrowserUrl(queryString);
}
```

Listing 12.11 »task-list.component.ts«: bisherige Implementierung der »findTasks«-Methode

Beim Drücken der ↵-Taste wird die Methode `findTasks` aufgerufen, die ihrerseits die Methode `findTasks` der `TaskService`-Klasse aufruft und das Ergebnis (ein `Observable` von `Task`-Objekten) an die `tasks$`-Variable bindet.

Bei der Umsetzung des Typeaheads werden Sie nun ein neues – Push-orientiertes – Konzept zur Behandlung von Nutzereingaben kennenlernen. Die grundlegende Idee dabei lautet wie folgt:

Nutzereingaben sind ein Strom von Daten. Kommen neue Daten in diesem Stream an, werden sie mithilfe von Operatoren zuerst in Anfragen an den Task-Service und anschließend in einen »Ergebnisstrom« verwandelt.

Der erste Schritt besteht also darin, die Werte des Eingabefeldes in einen Stream zu verwandeln. Wie Sie bereits aus Kapitel 9, »Reactive Forms: Formulare dynamisch in der Applikationslogik definieren«, wissen, bietet Angular Ihnen in diesem Zusammenhang die Möglichkeit, mithilfe der `valueChanges`-Eigenschaft der `FormControl`-Klasse ein `Observable` mit genau den gewünschten Daten zu erhalten:

```
tasks$: Observable<Task[]>;
searchTerm = new FormControl();
ngOnInit() {
  ...
  this.searchTerm.valueChanges.subscribe((value) => {
    console.log("Search Term:", value);
  });
}
```

Listing 12.12 »task-list.component.ts«: Erzeugung des »searchTerm«-Controls und Registrierung beim »valueChanges«-Observable

Ein Blick in die Developer-Konsole zeigt, dass die im Input-Feld eingegebenen Daten, wie in Abbildung 12.2 dargestellt, bereits wie gewünscht in einen Stream verwandelt werden.

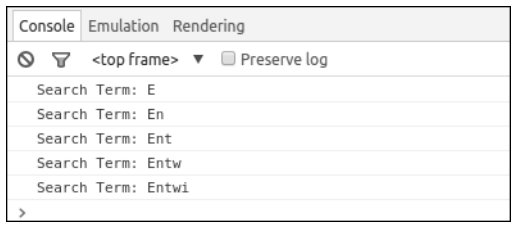

Abbildung 12.2 Ausgabe der eingegebenen Zeichen in der Developer-Konsole

Sie könnten nun innerhalb der `subscribe`-Methode einen Aufruf an den `TaskService` auslösen und somit die Task-Liste in Abhängigkeit vom eingegebenen Wert laden. Dies würde jedoch dazu führen, dass Ihr Server mit einer Vielzahl von unnötigen Anfragen überflutet werden würde. Bei der Entwicklung von Typeahead-Komponenten ist es somit üblich, die Anfrage an den Server erst dann zu stellen, wenn eine gewisse Zeit lang kein neuer Wert eingetippt wurde.

RxJS stellt für exakt diesen Use Case bereits einen komfortablen Operator zur Verfügung: Mithilfe des `debounceTime`-Operators teilen Sie RxJS mit, dass es eine gewisse Zeit lang warten soll, bis der Wert an den nächsten Operator weitergegeben wird. Kommt in der Zwischenzeit ein neuer Wert an, wird der alte Wert verworfen und nur der neue weitergeleitet. Listing 12.13 zeigt den Einsatz des `debounceTime`-Operators für die Typeahead-Implementierung:

```
import {debounceTime} from 'rxjs/operators';
...
this.searchTerm.valueChanges.pipe(
    debounceTime(400))
    .subscribe((value) => {
      console.log("Search Term:", value);
     });
```

Listing 12.13 Verwendung des »debounceTime«-Operators, um die Ausgabe zu verzögern

Durch das Einfügen des Operators wird die Log-Ausgabe nun nur noch dann ausgelöst, wenn der Nutzer 400 Millisekunden Pause zwischen den Tastaturanschlägen macht. Die Ausgaben in der Developer-Konsole werden somit stark reduziert (siehe Abbildung 12.3).

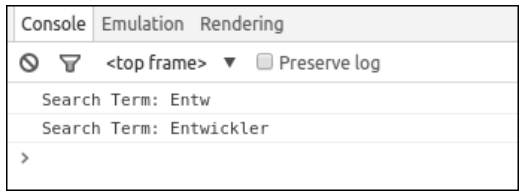

Abbildung 12.3 Verzögerte Ausgabe der Werte in der Developer-Konsole

Auf Basis der aktuellen Implementierung können Sie nun bereits das Laden der Tasks in die `subscribe`-Methode verlagern:

```
this.searchTerm.valueChanges.pipe
    (debounceTime(400))
    .subscribe((query) => {
      this.tasks$ = this.taskService.findTasks(query);
});
```

Listing 12.14 Anfrage an den »TaskService« innerhalb der »valueChanges«-Subscription

Immer wenn der Nutzer eine Tipp-Pause einlegt, wird eine neue Anfrage an den `TaskService` gestellt.

Die obige Implementierung funktioniert bereits einwandfrei. Da die `findTasks`-Methode jedoch selbst ein `Observable` zurückliefert, können Sie diese Implementierung noch deutlich eleganter gestalten. Die Idee ist, die vom `debounceTime`-Operator weitergeleiteten Werte direkt in Tasks umzuwandeln und sich anschließend (mithilfe der `AsyncPipe`) auf das Ergebnis zu subscriben. In den vorangegangenen Kapiteln haben Sie bereits einen Operator kennengelernt, der die Umwandlung von Eingangswerten in Ausgangswerte ermöglicht: den `map`-Operator. Eine erste Idee könnte also so aussehen, diesen Operator für die Umwandlung einzusetzen:

```
this.tasks$ = this.searchTerm.valueChanges.pipe(
    debounceTime(400),
    map(query => this.taskService.findTasks(query)));
```

Listing 12.15 Fehlerhafte Verwendung des »map«-Operators

Diese Implementierung führt jedoch zu einem Fehler. Ein Blick in die Developer-Konsole sollte das Bild aus Abbildung 12.4 zeigen.

Abbildung 12.4 Exception bei fehlerhafter Verwendung des »map«-Operators

Mehrere Operatoren auf einen Stream anwenden

Während Sie in den bisherigen Beispielen immer lediglich einen einzigen Operator auf ein `Observable` angewendet haben, zeigt Listing 12.15, dass Sie einen Stream mithilfe der `pipe`-Funktion auch mit mehreren Operatoren manipulieren können. So nimmt die Funktion eine beliebige Anzahl an Operatoren als Parameter entgegen, die dann – in der Reihenfolge, in der sie in der Parameterliste auftauchen – auf den Stream angewendet werden. Im Beispiel wird der Stream zunächst »gedebouncet«. Alle Werte, die es dann weiter in den Stream schaffen, werden anschließend über den `map`-Operator verändert.

Sie werden bei der Entwicklung von RxJS-Anwendungen immer wieder in die Situation kommen, dass die erzeugten Werte nicht denen entsprechen, die Sie eigentlich erwarten würden. Umso wichtiger ist es, eine Möglichkeit zu haben, die einzelnen Schritte zu debuggen. Für solche Aufgaben bietet sich der `tap`-Operator an. Der Operator nimmt selbst keine Manipulationen am Stream vor, sondern stellt Ihnen lediglich die Möglichkeit zur Verfügung, an beliebigen Stellen des Streams

Applikationslogik auszuführen. Listing 12.16 zeigt die entsprechende Verwendung zur Ausgabe der Stream-Werte:

```
this.tasks$ = this.searchTerm.valueChanges.pipe(
  debounceTime(400),
  map(query => this.taskService.findTasks(query)),
  tap(tasks => console.log('Tasks:', tasks))
);
```

Listing 12.16 Debugging von Observables mithilfe des »tap«-Operators

Über die Developer-Konsole können Sie nun das Problem herausfinden, das die Implementierung hat (siehe Abbildung 12.5).

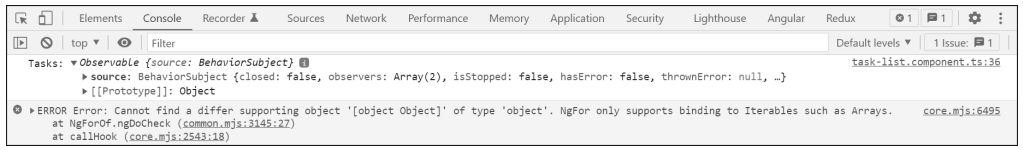

Abbildung 12.5 Ausgabe des Debug-Loggings

Da die `findTasks`-Methode selbst ein `Observable` zurückgibt, erzeugen Sie mit dem Aufruf

```
map(query => this.taskService.findTasks(query))
```

ein »Observable von Observables«. Die `AsyncPipe` versucht sich anschließend auf dieses `Observable` zu subscriben und erhält als Wert wiederum ein `Observable` – das Rendern der Liste schlägt somit mit der dargestellten Fehlermeldung fehl.

12.2.1 mergeMap: verschachtelte Observables verbinden

Da es sich beim Aufruf von Operationen, die wiederum ein Observable zurückgeben, um einen sehr üblichen Anwendungsfall in der reaktiven Anwendungsentwicklung handelt, stellt RxJS hierfür bereits einen passenden Operator bereit: Mithilfe des `mergeMap`-Operators können Sie innerhalb Ihrer Operatorenkette Aufrufe einbinden, die selbst ein `Observable` zurückgeben.

Anstatt anschließend das `Observable` selbst auf den Ergebnis-Stream zu leiten, leitet der `mergeMap`-Operator die *Werte* des inneren `Observable` auf den Stream des Haupt-Observables weiter. Bezogen auf das Typeahead-Beispiel ermöglicht der `mergeMap`-Operator es Ihnen somit, auf elegante Weise das vom `TaskService` bereitgestellte `Observable` zu integrieren:

```
this.tasks$ = this.searchTerm.valueChanges.pipe(
    debounceTime(400),
```



```
    mergeMap(query => this.taskService.findTasks(query)),
    tap(tasks => console.log('Tasks:', tasks)));
```

Listing 12.17 Verwendung des »mergeMap«-Operators zur Ausgabe des Task-Streams auf dem äußeren Datenstrom

Die Developer-Konsole zeigt nun das erwartete Ergebnis (siehe Abbildung 12.6).

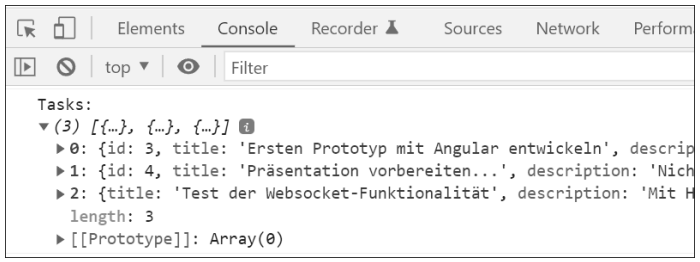

Abbildung 12.6 Korrektes Auslesen der Tasks mithilfe des »mergeMap«-Operators

Der `mergeMap`-Operator (in früheren RxJS-Versionen hieß er `flatMap`-Operator) ist definitiv einer der wichtigsten Operatoren bei der Entwicklung von reaktiven Anwendungen. So ermöglicht der Operator es Ihnen, sehr elegant verschiedene Bestandteile einer Applikation, die jeweils auf Observables basieren, miteinander zu verbinden. Zugegebenermaßen benötigt diese Art der Entwicklung anfangs etwas Umdenken – lassen Sie sich hiervon aber nicht abschrecken! Wenn Sie die Prinzipien einmal verinnerlicht haben, werden Sie nicht mehr anders entwickeln wollen.

12.2.2 switchMap: nur die aktuellsten Ergebnisse verarbeiten

Auch wenn die im vorigen Abschnitt vorgestellte Lösung bereits voll funktionsfähig ist, gibt es immer noch ein (nicht auf den ersten Blick offensichtliches) Problem mit der Implementierung: Dadurch, dass der `findTasks`-Aufruf asynchron arbeitet, kann es passieren, dass in Ihrer Oberfläche Werte aus »alten« Aufrufen dargestellt werden. Stellen Sie sich folgendes Szenario vor:

1. Der Nutzer will nach allen Tasks mit dem Begriff »Entwickler« suchen. Er tippt zunächst »Ent« und macht eine kurze Pause. Anfrage 1 wird mit der Query »Ent« an den Server gesendet.
2. Der Nutzer tippt weiter und vervollständigt die Suche zu »Entwickler«. Anfrage 2 wird mit dem Wert »Entwickler« abgesendet.
3. Da die zweite Anfrage weniger Ergebnisse liefert, kann sie vom Server schneller beantwortet werden. Nach 200 ms kommt die Antwort vom Server zurück und die Tabelle wird mit den gewünschten Ergebnissen der Anfrage »Entwickler« gefüllt.

4. Nach 500 ms kommt schließlich die Antwort auf die erste Anfrage zurück. Die Tabelle wird nun mit den Ergebnissen der Anfrage »Ent« gefüllt, enthält also beispielsweise auch Einträge mit dem Suchwort »Entwurf« oder Ähnlichem.

RxJS bietet Ihnen für genau dieses Szenario ebenfalls einen vorgefertigten Operator an: Der `switchMap`-Operator leitet nur Ergebnisse der letzten Anfrage weiter. Im obigen Ablauf hätte der Operator also gemerkt, dass die in Schritt 4 einlaufenden Ergebnisse »alt« sind, und hätte die Ergebnisse verworfen. Listing 12.18 zeigt die korrekte Verwendung des `switchMap`-Operators:

```
this.tasks$ = this.searchTerm.valueChanges.pipe(
    debounceTime(400),
    switchMap(query => this.taskService.findTasks(query))
);
```

Listing 12.18 Verwendung des »switchMap«-Operators, um »alte« Anfragen zu verwerfen

Die Ergebnisliste zeigt nun immer das Ergebnis des letzten Aufrufs. Mit drei Zeilen Code haben Sie eine voll funktionale Typeahead-Suche implementiert!

12.2.3 merge: mehrere Streams vereinen

In Abschnitt 12.2.2 haben Sie die Task-Suche über das Eingabefeld auf reaktive Art und Weise umgesetzt. Außer der Eingabe in das Suchfeld haben Sie zum aktuellen Zeitpunkt aber immer noch eine weitere Stelle im Quellcode, die das Laden eines Tasks auslöst, nämlich die Auswertung der vom Routing-Framework bereitgestellten Query-Parameter:

```
this.route.queryParams.subscribe(params => {
  const query = decodeURI(params['query'] ?? '');
  this.searchTerm.setValue(query);
  this.tasks$ = this.taskService.findTasks(query);
});
```

Listing 12.19 »task-list.component.ts«: bisherige Auswertung der Query-Parameter

Architektonisch betrachtet, haben Sie somit im Grunde genommen zwei *Datenquellen*, die Anfragen an den `TaskService` erzeugen: den Suchfeld-Stream und den Query-Parameter-Stream. Kommt ein neuer Wert über den Suchfeld-Stream an, möchten Sie zusätzlich zur Task-Suche automatisch die URL anpassen (`adjustBrowserUrl`); und bei Werten aus dem Query-Parameter-Stream soll das Suchfeld mit dem entsprechenden Wert befüllt werden (`searchTerm.setValue`).

Die Modellierung von reaktiven Applikationen basiert genau auf diesem Denkmuster: Der Zustand Ihrer Anwendung wird im Endeffekt durch die Entgegennahme und Transformation von Daten (den sogenannten Datenfluss) bestimmt.

Schauen Sie sich, um dieses Konzept besser zu verstehen, zunächst einmal die Definition der folgenden beiden Streams an:

```
const paramsStream$ = this.route.queryParams.pipe(
  map(params => decodeURI(params['query'] ?? '')),
  tap(query => this.searchTerm.setValue(query))
);

const searchTermStream$ = this.searchTerm.valueChanges.pipe(
  debounceTime(400),
  tap(query => this.adjustBrowserUrl(query))
);
```

Listing 12.20 »task-list.component.ts«: die beiden Datenquellen für die Task-Suche

Wie im vorigen Abschnitt beschrieben, enthält der `paramsStream$` die Werte, die über die Browser-URL in die Anwendung gegeben werden. Kommt ein neuer Wert an, wird automatisch das Suchfeld mit dem Wert befüllt. Äquivalent dazu repräsentiert der `searchTermStream$` die Werte, die über das Suchfeld in die Anwendung gegeben werden.

Und nun wird es interessant: Indem Sie die beiden Datenströme verbinden, erhalten Sie einen neuen Datenstrom, der sowohl die Werte aus der Browser-URL als auch die Werte aus dem Suchfeld enthält. RxJS stellt Ihnen hierfür den Erzeugungsoperator `merge` zur Verfügung, der aus zwei bestehenden Datenströmen einen einzigen kombinierten Stream erzeugt:

```
import {merge, Observable} from 'rxjs';
...
this.tasks$ = merge(paramsStream$, searchTermStream$).pipe(
  distinctUntilChanged(),
  switchMap(query =>  this.taskService.findTasks(query))
);
```

Listing 12.21 »tasks-list.component.ts«: Auslösen der Task-Suche auf dem vereinigten Query-Stream

Über den `merge`-Operator erzeugen Sie zunächst den neuen vereinigten Stream. Der anschließende `distinctUntilChanged`-Operator hat hier die Aufgabe, Werte nur bei einer Änderung weiterzuleiten. Haben Sie beispielsweise bereits über die Browser-URL nach dem Wort »Entwickler« gesucht und würden Sie anschließend einen weiteren Query-Parameter zur URL hinzufügen, dann würde der `paramsStream$` einen neuen Wert in den kombinierten Stream schicken. Da sich das Suchwort aber nicht geändert hat, würde keine erneute Task-Suche ausgelöst werden. Im letzten Schritt

wird nun lediglich noch das Ergebnis der eigentlichen Suche über den `switchMap`-Operator auf den äußeren Stream weitergeleitet.

Fertig! Listing 12.22 zeigt noch einmal die gesamte Implementierung des reaktiven Datenstroms zum Laden von Tasks über den `TaskService`:

```
const paramsStream$ = this.route.queryParams.pipe(
  map(params => decodeURI(params['query'] ?? '')),
  tap(query => this.searchTerm.setValue(query)));

const searchTermStream$ = this.searchTerm.valueChanges.pipe(
  debounceTime(400),
  tap(query => this.adjustBrowserUrl(query)));

this.tasks$ = merge(paramsStream$, searchTermStream$).pipe(
  distinctUntilChanged(),
  switchMap(query =>  this.taskService.findTasks(query)));
```

Listing 12.22 »task-list.component.ts«: komplette Umsetzung der Datenanforderung über den »TaskService«

Mit gerade einmal neun Zeilen Quellcode haben Sie ein Typeahead-Suchfeld implementiert, die Auswertung von Browser-URL-Parametern umgesetzt und eine einzelne zentrale Stelle für die Anforderung von Tasks aus dem `TaskService` geschaffen. Ein Blick in Listing 12.22 zeigt außerdem, dass sich die Datenflüsse Ihrer Applikation sehr sprechend im implementierten Quellcode widerspiegeln!

Wenn Sie nun die Oberfläche öffnen und die Browser-URL verändern oder Suchbegriffe in das Suchfeld tippen, werden Sie feststellen, dass die Eingangsdaten wie erwartet durch die Streams geleitet werden und im Endeffekt zu einer Darstellung der Tasks in der Oberfläche führen (siehe Abbildung 12.7).

Das »Single Source of Truth«-Prinzip

Sollten Sie sich bereits im Internet über RxJS und reaktive Programmierung informiert haben, so werden Sie mit Sicherheit auch schon über eines *der* Marketing-Buzzwords der Bibliothek gestolpert sein: über das *Single Source of Truth*-Prinzip. Wenn Sie sich damals gefragt haben, was es damit auf sich hat: Sie haben es gerade kennengelernt!

Die einzige Quelle für Tasks ist das `tasks$`-Observable. Wollen Sie Tasks auf eine andere Art und Weise laden, so geschieht dies immer dadurch, dass Sie diese durch den entsprechenden Stream schicken. Im Gegensatz dazu hätten Sie in einer klassischen »Pull-orientierten« Architektur mindestens zwei Aufrufe an den `TaskService` implementiert: einen Aufruf, der bei einer Suche über das Suchfeld durchgeführt wird, und einen Aufruf, der als Reaktion auf die Query-Parameter-URL erfolgt.

Durch die Bündelung der Tasks in einen einzigen Stream wissen Sie nun immer genau, wo Sie nach gegebenenfalls auftretenden Fehlern suchen müssen. Neben der einfacheren Fehlersuche besitzt dieser Ansatz noch eine Vielzahl von weiteren Vorteilen, die Sie im folgenden Abschnitt noch genauer kennenlernen werden.

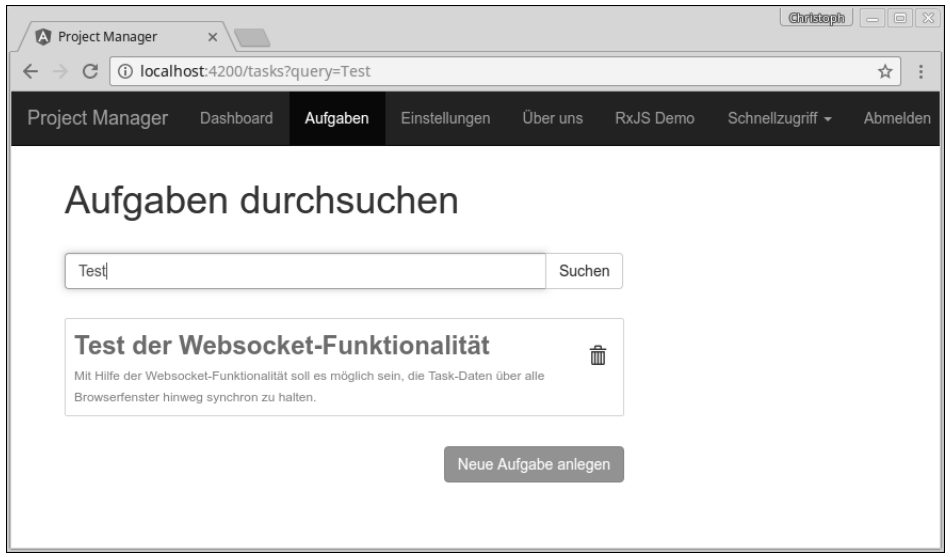

Abbildung 12.7 Die vollständige Typeahead-Implementierung in Aktion

12.3 Reaktive Datenarchitekturen in Angular-Applikationen

Sie haben jetzt bereits einen Eindruck davon bekommen, wie Sie mit RxJS sehr elegant auf User-Eingaben reagieren und diese in Datenströme verwandeln können. In diesem Abschnitt werden Sie nun noch einen großen Schritt weiter gehen.

Sie werden nicht mehr »nur« die Verwaltung der vom Controller benötigten Daten über Observables kapseln, sondern die gesamte Datenarchitektur Ihrer Anwendung reaktiv gestalten. Insbesondere bei der Entwicklung von Applikationen mit hohen Performance-Anforderungen oder bei der Anbindung alternativer Datenquellen (wie z. B. Websockets) bietet Ihnen diese Art der Entwicklung bedeutende Vorteile.

Lassen Sie mich zunächst einmal das »Problem« der bisherigen Architektur erläutern. Stellen Sie sich beispielsweise vor, Sie wollten die Schnellansicht eines Tasks, die ich in Kapitel 10, »Routing: Navigation innerhalb der Anwendung«, vorgestellt habe, so erweitern, dass sie die Möglichkeit bietet, die wichtigsten Bestandteile des Tasks so, wie es in Abbildung 12.8 dargestellt ist, *on the fly* zu bearbeiten und zu speichern.

Sollten Sie bereits einmal mit dem Tool *JIRA* in Kontakt gekommen sein, werden Sie diese Funktion sicher kennen: Sobald ein Task in der Liste markiert wird, wird auf der rechten Seite die Schnellansicht dargestellt. In dem Beispiel, das in diesem Abschnitt implementiert wird, bietet diese Schnellansicht nun die Möglichkeit, den Titel, die Beschreibung und den Status eines Tasks zu ändern.

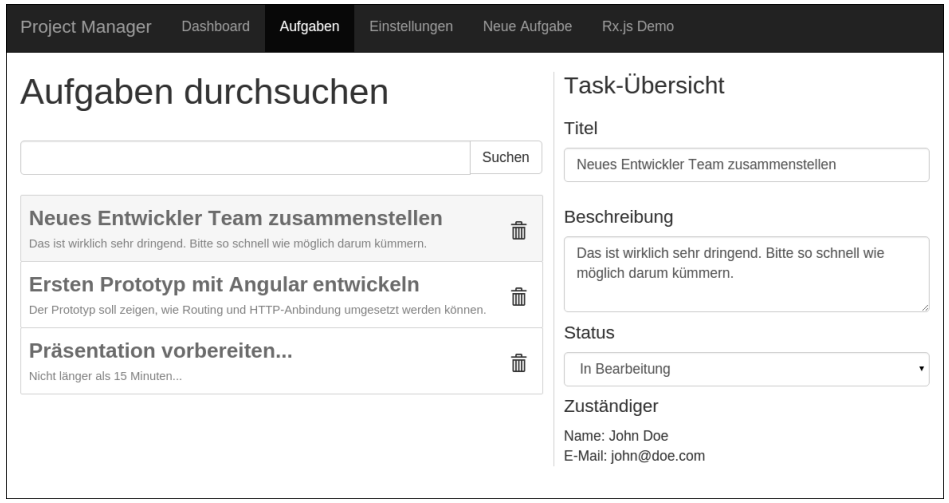

Abbildung 12.8 »Quick-Edit-Ansicht« in der Task-Übersicht

Im Gegensatz zur regulären Editierfunktion soll der Task in dieser Ansicht sofort bei der Bestätigung der Eingabe durch die ↵-Taste gespeichert werden. Nach dem erfolgreichen Speichern wird des Weiteren für zwei Sekunden eine Erfolgsnachricht eingeblendet. Listing 12.23 und Listing 12.24 zeigen die hierfür zuständigen Ausschnitte der angepassten `TaskOverviewComponent`:

```
<h3>Task-Übersicht
  <span *ngIf="showSuccessLabel"
        class="label label-success pull-right label-small">
      Erfolgreich gespeichert
  </span>
</h3>
...
<div class="form-group">
  <label for="title">Titel</label>
  <input type="text" id="title" class="form-control"
         [(ngModel)]="task.title"
         (keyup.enter)="saveTask()"/>
</div>
...
```

Listing 12.23 »task-overview.component.html«: Darstellung der Erfolgsnachricht und des »Form«-Elements zum Speichern des Tasks


```
saveTask(): void {
  this.taskService.saveTask(this.task).subscribe(task => {
    this.task = task;
    this.showSuccessLabel = true;
    setTimeout(() => {
      this.showSuccessLabel = false;
    }, 2000)
  });
}
```

Listing 12.24 »task-overview.component.ts«: Methode zum Speichern des Tasks

Das Problem dieser Implementierung wird bei einem erneuten Blick in die Oberfläche deutlich: Wie in Abbildung 12.9 dargestellt, wird der Task zwar nach Betätigung der ↵-Taste gespeichert – die Task-Liste auf der linken Seite bekommt von dieser Änderung jedoch nichts mit und zeigt weiterhin den alten Wert des Titels an.

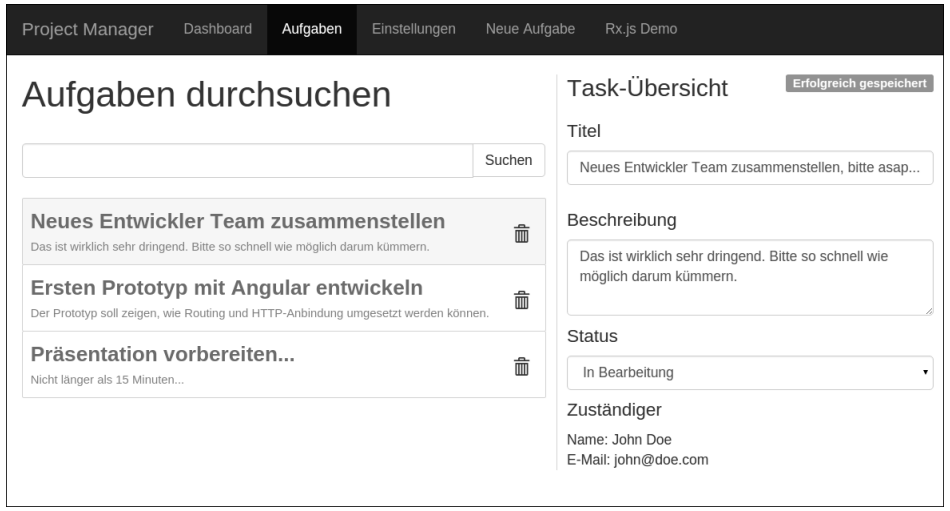

Abbildung 12.9 Keine Aktualisierung der Task-Liste

Sie benötigen also eine Möglichkeit, innerhalb der Liste darüber informiert zu werden, wenn sich Änderungen an den Tasks ergeben.

12.3.1 Shared Services: der erste Schritt in die richtige Richtung

Eine (durchaus übliche) Lösung für diese Aufgabenstellung besteht darin, im Fall einer Änderung an den Tasks im `TaskService` ein Event auszulösen, das interessierte Teilnehmer darüber informiert, dass Änderungen an den Tasks eingetreten sind. Der `TaskService` ist somit dafür zuständig, als »zentrale Stelle« die Änderungen von Tasks zu managen:

```
export class TaskService {
  ...
  taskSaved$ = new Subject<Task>();
  saveTask(task: Task): Observable<Task> {
    ...
    return this.http.request<Task>(method, `${BASE_URL}/${task.id ?? ''}`,
    { body: task }).pipe(
      tap(savedTask => this.taskSaved$.next(savedTask))
    );
  }
}
```

Listing 12.25 »task.service.ts«: Auslösen des »taskSaved«-Events innerhalb der »saveTask«-Methode im »TaskService«

Wie Sie sehen, verwendet Listing 12.25 die in Abschnitt 12.1.3 vorgestellte Klasse `Subject` zur Information der anderen Teilnehmer.

Innerhalb der `TaskListComponent` können Sie das `taskSaved$`-Subject nun dazu als Kriterium zum Neuladen der Tasks aufnehmen:

```
export class TaskListComponent {
  ...
  ngOnInit() {
    ...
    this.tasks$ = merge(
      paramsStream$,
      searchTermStream$,
      this.taskService.taskSaved$
    ).pipe(
      distinctUntilChanged(),
      switchMap(() => this.taskService.findTasks(this.searchTerm.value))
    );
  }
}
```

Listing 12.26 »task-list.component.ts«: Die Tasks werden neu geladen, wenn ein Wert auf dem »taskSaved$«-Stream eintrifft.

Beachten Sie in Listing 12.26 auch, dass Sie nun innerhalb des `switchMap`-Operators direkt auf die `value`-Eigenschaft des `searchTerm`-Feldes zugreifen müssen, da der `taskSaved$`-Stream nicht den verwendeten Suchbegriff, sondern den gespeicherten Task enthält:

```
switchMap(() => this.taskService.findTasks(this.searchTerm.value))
```


Abbildung 12.10 zeigt, dass die Task-Liste beim Speichern eines Tasks nun wie gewünscht aktualisiert wird.

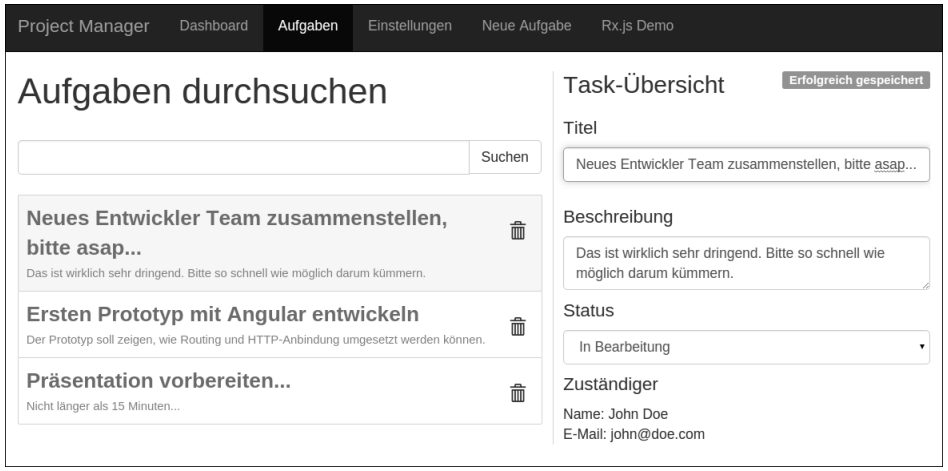

Abbildung 12.10 Korrekte Aktualisierung der Task-Liste

Das Problem mehrerer Datenquellen

Auch wenn die vorgestellte Umsetzung nun dafür sorgt, dass ein Speichern aus der Schnellansicht heraus ebenfalls die Task-Liste aktualisiert, gibt es leider immer noch eine (architektonische) Unschönheit im Quellcode: Die `TaskListComponent` muss zum aktuellen Zeitpunkt ganz genau wissen, wann sie die Daten aus dem `TaskService` neu anfordern muss.

Im vorliegenden Fall mag dies noch einigermaßen überschaubar sein. Durch die Anbindung eines Websocket-Mechanismus (siehe Abschnitt 12.4) hätten Sie aber bereits ein viertes Kriterium zum Neuladen der Tasks. Und damit nicht genug: Im nächsten Meeting kommt Ihr Produktmanager auf die Idee, die Anzahl der Tasks, die sich aktuell in Bearbeitung befinden, im Header der Seite darzustellen.

Bei der Implementierung dieses Features stoßen Sie auf die gleichen Probleme wie zuvor: Zunächst müssen Sie die Zahl initial errechnen und dann jeweils auf alle möglichen Änderungen an den Tasks reagieren. Schleicht sich hierbei ein Fehler ein, laufen die Zahlen im Header und in der Liste auseinander ...

Ich denke, Sie sehen, welche Probleme die Anbindung verschiedener Datenquellen mit sich bringen kann.

12.3.2 Die neue Datenarchitektur: »Push« statt »Pull«

Das grundlegende Problem der bisherigen Lösung liegt vereinfacht gesagt darin, dass sich jede Komponente der Anwendung selbstständig die Daten aus dem `TaskService` »zieht«. Jede Komponente muss somit wissen, wann es Änderungen gibt, und diese

dann aktiv anfordern. Der in diesem Abschnitt vorgestellte Lösungsansatz geht hier einen radikal anderen Weg: Anstatt sich beim Auftreten eines Events selbst darum zu kümmern, die Daten abzuholen, meldet sich eine Komponente *einmalig* beim Service an und registriert sich dort für Änderungen an den Daten. In der Folge versorgt der Service alle angemeldeten Komponenten über einen *Push-Mechanismus* aktiv mit den aktuellen Werten. Die tatsächliche Anforderung von Daten geschieht nun durch das Triggern einer *Aktion*.

Möchte die Komponente beispielsweise neue Daten vom Service empfangen (etwa weil der Benutzer etwas in das Suchfeld eingegeben hat), so triggert sie beim Service die `LOAD`-Aktion. Nach Abschluss der Aktion versorgt der Service alle registrierten Komponenten mit den aktualisierten Daten.

Beim ersten Lesen klingt dies vermutlich sehr abstrakt. Ich möchte Ihnen die Unterschiede deshalb auch grafisch verdeutlichen. Abbildung 12.11 zeigt zunächst die klassische Architektur.

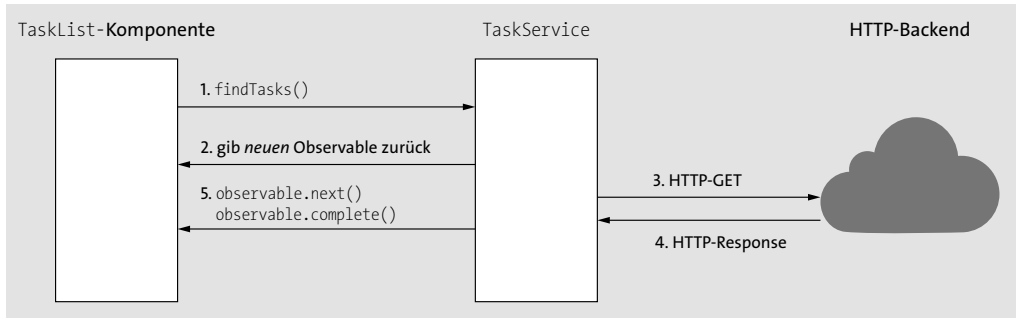

Abbildung 12.11 Klassische Datenarchitektur

Der Service dient in diesem Fall lediglich zur Kapselung der Datenoperationen. Möchte die Komponente neue Werte laden, so fordert sie vom Service einen neuen `Observable`-Stream an, der nach dem erfolgreichen Abruf der Daten vom HTTP-Backend die Werte an die Komponente liefert und anschließend beendet wird.

Der neue Push-orientierte Ansatz sieht auf den ersten Blick etwas komplizierter aus (siehe Abbildung 12.12).

Der zentrale Unterschied dieses Ansatzes besteht darin, dass die Komponente nicht bei jedem Laden von Daten ein neues `Observable` vom Service erhält. Vielmehr registriert sich die Komponente einmalig beim sogenannten *Store* für neue Werte. (Das Konzept des Stores werde ich Ihnen in Kürze vorstellen.)

Anschließend kann die Komponente verschiedene *Aktionen* – z. B. die bereits angesprochene `LOAD`-Aktion – am Service auslösen. Diese Aktion sorgt dann beispielsweise dafür, dass die angeforderten Daten vom HTTP-Backend geladen und anschließend

in den Store geschrieben werden. Im letzten Schritt »pusht« der Store die geladenen Daten in die Komponente hinein.

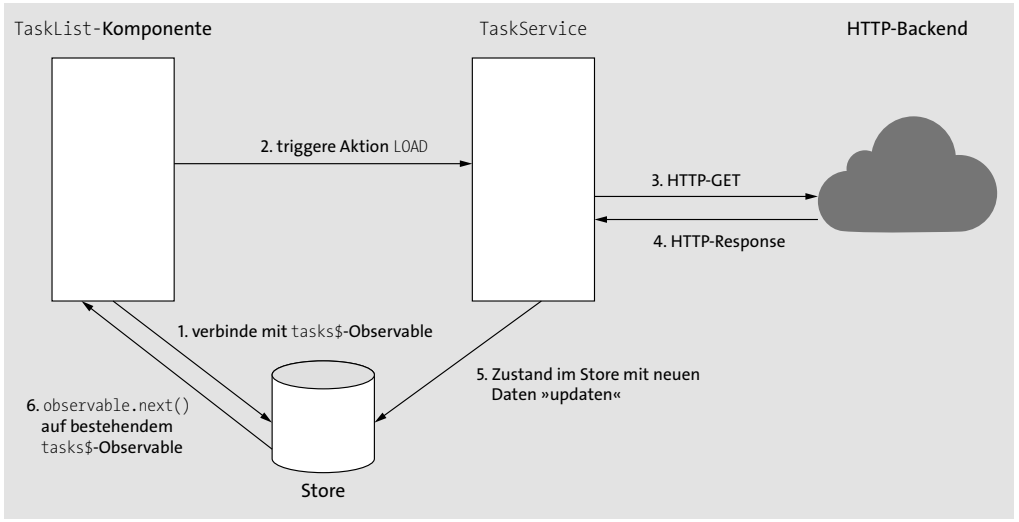

Abbildung 12.12 Die neue »Push-orientierte« Datenarchitektur

Der große Vorteil dieses Konzepts wird deutlich, wenn eine zusätzliche Komponente (z. B. die `TaskOverviewComponent`) ins Spiel kommt. Abbildung 12.13 demonstriert diesen Ablauf aufbauend auf dem zuvor beschriebenen Szenario.

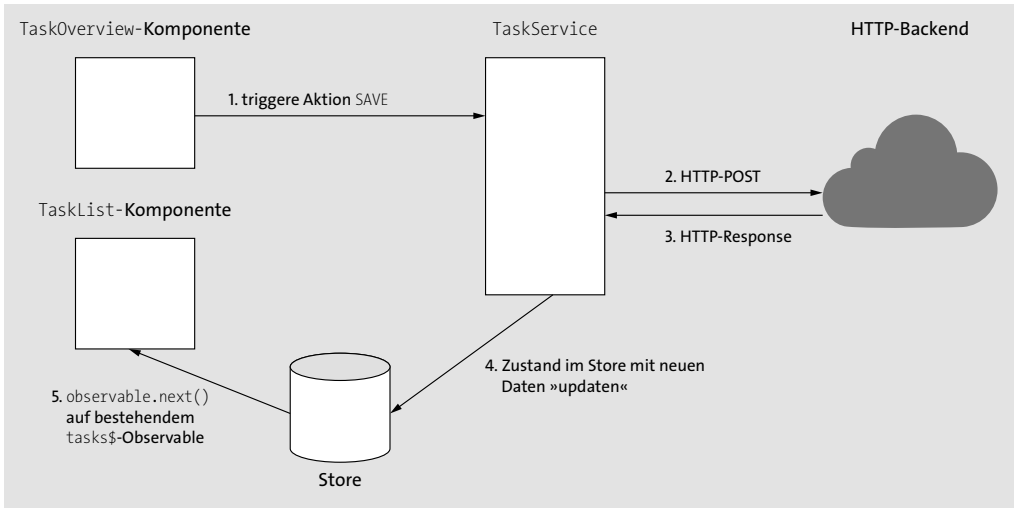

Abbildung 12.13 Speichern aus der »TaskOverview«-Komponente heraus

Das Schaubild zeigt die Aktion des Speicherns aus der Schnellansicht heraus, die ich im vorigen Abschnitt beschrieben habe. Im Fall der Push-Architektur löst die `TaskOver-`

viewComponent lediglich die SAVE-Aktion aus. Der TaskService speichert anschließend den Task über das HTTP-Backend.

Nun kommt der entscheidende Punkt: Nach dem Speichern im Backend sorgt der Service dafür, dass der aktualisierte Task in die im Store enthaltene Liste integriert wird. (In Wirklichkeit wird die Liste im Store ausgetauscht, aber dazu gleich mehr.) Nach dem Aktualisieren der Liste werden alle angemeldeten Subscriber (also auch die TaskListComponent) mit der neuen Task-Liste versorgt – die Ansicht wird automatisch aktualisiert. Da dieser Punkt so zentral für das beschriebene Konzept ist, noch einmal mit anderen Worten:

Wenn eine Aktion dafür sorgt, dass sich der Zustand (State) des Stores ändert, so werden alle angemeldeten Komponenten automatisch über diese Änderungen informiert und aktualisiert.

12.3.3 Umsetzung des neuen Konzepts in Angular

Das vorgestellte Konzept löst also viele der zuvor beschriebenen Probleme der Datenverwaltung. Doch wie kann dieser Ansatz nun konkret in Ihre Angular-Anwendung integriert werden?

Schauen Sie sich hierfür zunächst den folgenden Ausschnitt der notwendigen Erweiterungen am TaskService an. Die Implementierungsdetails der Klasse TaskStore werden Sie dann im Anschluss kennenlernen.

```
import { TaskStore } from '../stores/task.store';
import { ActionType, ADD, EDIT, LOAD, REMOVE } from '../stores/action';
...
export class TaskService {
  selectTasks(): Observable<Task[]> {
    return this.taskStore.selectItems();
  }
  ...
  findTasks(query = '', sort = 'id', order = 'ASC'): Observable<Task[]> {
    ...
    return this.http.get<Task[]>(BASE_URL, {params: searchParams})
      .pipe(tap((tasks) => {
        this.taskStore.dispatch({type: LOAD, data: tasks});
      }));
  }
  saveTask(task: Task): Observable<Task> {
    ...
    return this.http.request<Task>(method, `${BASE_URL}/${task.id ?? ''}`, {
         body: task
       })
      .pipe(tap(savedTask => {
        const actionType = task.id ? EDIT : ADD;
        this.taskStore.dispatch({type : actionType, data: savedTask});
      }));
  }
  ...
}
```

Listing 12.27 »task.service.ts«: Integration des »TaskStore« in die »TaskService«-Klasse

Listing 12.27 zeigt exemplarisch die Implementierungen der Methoden selectTasks, findTasks und saveTask. Wie in Abschnitt 12.3.2 beschrieben, wird hier zusätzlich zum Service die neue Klasse TaskStore integriert, die die zentrale Verwaltung der Tasks übernimmt. Im Fall eines Service-Aufrufs wird nach dem Erhalt der HTTP-Antwort eine Action »dispatcht«, die dafür sorgt, dass der Store einen neuen Zustand erhält. Beachten Sie dabei, dass eine Action im Endeffekt durch ein einfaches JavaScript-Objekt realisiert wird:

```
this.taskStore.dispatch({type: LOAD, data: tasks});
```

Listing 12.28 zeigt die Implementierung des zugehörigen Action-Interfaces:

```
export const LOAD = 'LOAD';
export const ADD = 'ADD';
export const EDIT = 'EDIT';
export const REMOVE = 'REMOVE';

export type ActionType =
 typeof  LOAD | typeof  ADD | typeof  EDIT | typeof  REMOVE;

export interface Action {
  type: ActionType;
  data: any;
}
```

Listing 12.28 »action.ts«: Definition des »Action«-Interface und der zur Verfügung stehenden Action-Typen

Wie Sie sehen, sind die zur Verfügung stehenden Aktionen ebenfalls in der Datei *action.ts* definiert. Interessant ist hier außerdem die Definition des ActionType-Typs:

```
export type ActionType =
 typeof  LOAD | typeof  ADD | typeof  EDIT | typeof  REMOVE;
```

Hierbei handelt es sich um einen sogenannten String-Literal-Typ. So sorgt die hier dargestellte Definition dafür, dass Sie der `type`-Eigenschaft eines `Action`-Objekts lediglich die zuvor definierten Werte zuweisen können. Sie lernen dieses TypeScript-Sprachkonstrukt im Detail in Abschnitt B.6 kennen.

Die Implementierung der `TaskStore`-Klasse wertet diese Aktionen entsprechend aus und informiert angemeldete Subscriber über Änderungen. Listing 12.29 zeigt die komplette Implementierung des `TaskStore`:

```
import {BehaviorSubject} from 'rxjs';
import {Task} from '../../models/model-interfaces';

export class TaskStore {
  private tasks: Task[] = [];
  private items$ = new BehaviorSubject<Task[]>([]);
  selectItems(): Observable<Task[]> {
    return this.items$.asObservable();
  }
  dispatch(action: Action) {
    this.tasks = this.reduce(this.tasks, action);
    this.items$.next(this.tasks);
  }
  private reduce(tasks: Task[], action) {
    switch (action.type) {
      case LOAD:
        return [...action.data];
      case ADD:
        return [...tasks, action.data];
      case EDIT:
        return tasks.map(task => {
          const editedTask = action.data;
          if (task.id !== editedTask.id){
            return task;
          }
          return editedTask;
        });
      case REMOVE:
        return tasks.filter(task => task.id !== action.data.id);
      default:
        return tasks;
    }
  }
}
```

Listing 12.29 »task.store.ts«: Implementierung des »TaskStore«

Konzentrieren Sie sich zunächst auf den folgenden Ausschnitt der Klasse:

```
export class TaskStore {
  private tasks: Task[] = [];
  private items$ = new BehaviorSubject<Task[]>([]);
  ...
  dispatch(action: Action) {
    this.tasks = this.reduce(this.tasks, action);
    this.items$.next(this.tasks);
  }
  ...
}
```

Listing 12.30 Verarbeitung der Aktion und Information der angemeldeten Subscriber

Die `TaskStore`-Klasse verwaltet den aktuellen Status hier über ein Objekt der Klasse `BehaviorSubject`. Hierbei handelt es sich um ein spezielles `Subject` der RxJS-Bibliothek, das sich insbesondere für die Implementierung von Observables eignet, die im Laufe der Zeit neue Subscriber erhalten. Die Besonderheit eines `BehaviorSubject` besteht darin, dass ein neuer `Subscriber` nach der Anmeldung automatisch den letzten im `Observable` verfügbaren Wert zugestellt bekommt. `BehaviorSubject`s eignen sich somit hervorragend für die Verteilung des aktuellen Zustands an mehrere Subscriber.

Die `dispatch`-Methode ist die einzige öffentliche Schnittstelle zur Veränderung der Store-Daten. Die eigentliche Manipulation der Daten geschieht dabei über eine sogenannte *Reducer Function*. Die Funktion nimmt als Parameter den bisherigen *State* (Zustand) sowie die auszuführende Aktion entgegen. Als Rückgabe liefert die Funktion den neuen State. Nach der Berechnung des States wird dieser schließlich über das `BehaviorSubject` an alle interessierten Komponenten versendet.

Kurzexkurs: Redux

Bei der hier vorgestellten Lösung handelt es sich um eine Kombination aus RxJS-Funktionalitäten in Verbindung mit Ideen aus dem JavaScript-Framework *Redux*. Redux ist ein (Achtung: Buzzword-Alarm) *State-Container*, der auf dem *Flux*-Pattern basiert. Das Framework wurde von Dan Abramov entwickelt und setzt im Wesentlichen die folgenden Ideen um:

1. Alle Daten (der *State*) der Applikation liegen in einem zentralen JavaScript-Objekt (dem *Store*) vor.
2. Dieser Store ist ein »Read-only«-Speicher. Änderungen sorgen immer dafür, dass eine neue Instanz des aktuellen States erstellt wird.

3. Der einzige Weg, um Änderungen am State vorzunehmen, besteht darin, *Actions* an den Store zu übergeben Ein direkter Schreibzugriff auf die Daten ist nicht möglich.
4. Die sogenannte *Reducer*-Funktion erzeugt aus den »alten Daten« und der Aktion den neuen State.
5. Die Reducer-Funktion muss als *Pure Function* implementiert sein, darf also keine Seiteneffekte besitzen.

Auch wenn die momentane Implementierung nicht den kompletten Zustand der gesamten Applikation in einem einzigen Objekt vorhält, sind die grundsätzlichen Ideen (insbesondere der *Immutable*-State) in Verbindung mit Angular-Funktionalitäten sehr mächtig. Schauen Sie sich also als Nächstes die Umsetzung der *Reducer*-Funktionalität an:

```
private reduce(tasks: Task[], action: Action) {
  switch (action.type) {
    case LOAD:
      return [...action.data];
    case ADD:
      return [...tasks, action.data];
    case EDIT:
      return tasks.map(task => {
        const editedTask = action.data;
        if (task.id !== editedTask.id){
          return task;
        }
        return editedTask;
      });
    case REMOVE:
      return tasks.filter(task => task.id !== action.data.id);
    default:
      return tasks;
  }
}
```

Listing 12.31 »task.store.ts«: die »Reducer«-Funktion des »TaskStore«

Der wichtigste Grundsatz lautet hier, dass jede Aktion eine *neue Liste* mit Tasks erzeugt und diese zurückgibt. Der aktuelle Zustand wird nicht verändert.

Die LOAD-Action

Die Implementierung der `LOAD`-Action ist hierbei trivial. So sorgt diese Aktion dafür, dass die momentan im Store vorhandenen Daten durch die übergebenen Daten

ersetzt werden. Die Reducer-Funktion gibt also lediglich eine Kopie der übergebenen Daten zurück.

Sie fertigen diese Kopie dabei mithilfe des ECMAScript-2015-*Spread-Operators* an (siehe Anhang A, »ECMAScript 2015 (and beyond)«). Der Ausschnitt

```
case LOAD:
  return [...action.data];
```

sorgt also dafür, dass ein neues Array erzeugt wird, das alle Elemente des Arrays `action.data` enthält. Durch den Aufruf der `dispatch`-Funktion aus dem `TaskService` heraus wird der Store somit mit den Ergebnissen der `findTasks`-Methode gefüllt:

```
findTasks(query = '', sort = 'id', order = 'ASC') {
  ...
  this.taskStore.dispatch({type: LOAD, data: tasks});
}
```


Die ADD-Action

Die `ADD`-Action ist ähnlich trivial. Sie erzeugt, ebenfalls mithilfe des Spread-Operators, ein neues Array, das zunächst alle bisherigen Werte und zusätzlich das in `action.data` übergebene Element enthält:

```
case ADD:
  return [...tasks, action.data];
```

Die EDIT-Action

Die `EDIT`-Action ist die aufwendigste Aktion:

```
case EDIT:
  return tasks.map(task => {
    const editedTask = action.data;
    if (task.id !== editedTask.id){
      return task;
    }
    return editedTask;
  });
```

Mithilfe der `Array.map`-Funktion wird ein neues Array erstellt, bei dem der editierte Task im Ursprungs-Array ausgetauscht wird. Alle weiteren Tasks werden unverändert in die neue Liste übernommen.

Die Verwendung der `ADD`- und `EDIT`-Aktion können Sie sich zur Verdeutlichung noch einmal im `TaskService` in der Methode `saveTask` anschauen. Hier wird – abhängig davon, ob der zu speichernde Task bereits eine `id` besitzt oder nicht – entweder die `ADD`- oder die `EDIT`-Action an den Store übergeben:

```
saveTask(task: Task): Observable<Task> {
  ...
  .pipe(tap(savedTask => {
    const actionType: ActionType = task.id ? EDIT : ADD;
    this.taskStore.dispatch({type : actionType, data: savedTask});
  }));
}
```

Die REMOVE-Action

Die `REMOVE`-Action ist wieder sehr leicht verständlich. Mithilfe der `Array.filter`-Methode wird hier ein neues Array erzeugt, in dem der gelöschte Task ausgefiltert wurde:

```
case REMOVE:
  return tasks.filter(task => task.id !== action.data.id);
```

Innerhalb der `deleteTask`-Methode des `TaskService` erfolgt der Aufruf wie erwartet nach der Rückkehr des HTTP-Aufrufs:

```
deleteTask(task: Task) {
  return this.http.delete<Task>(`${BASE_URL}/${task.id}`)
      .pipe(tap(() => {
        this.taskStore.dispatch({type: REMOVE, data: task});
  }))
}
```

Die default-Action

Die Implementierung eines Default-Verhaltens innerhalb der Reducer-Funktion ist insofern wichtig, als dass der Rückgabewert der `reduce`-Funktion ohne weitere Überprüfung als »neuer State« hinterlegt wird. Wird der Reducer also mit einer unbekannten Action aufgerufen, muss sichergestellt sein, dass der Store dadurch nicht aus Versehen gelöscht wird. In diesem Fall wird somit einfach der aktuelle Status zurückgegeben:

```
default:
  return tasks;
```

Generische Implementierung der Store-Komponente

Bei genauerer Betrachtung der TaskStore-Implementierung werden Sie feststellen, dass der Code nahezu keine fachliche Abhängigkeit zur `Task`-Klasse hat. Anders ausgedrückt: Ein weiterer Store (z. B. zur Verwaltung von Kontakten) würde zu 95 % aus Copy&Paste-Code bestehen. TypeScript bietet für diesen Fall die Verwendung von *Generics* an.

Sollten Sie sich für das Thema interessieren, biete ich Ihnen in Anhang B, »Typsicheres JavaScript mit TypeScript«, eine detaillierte Beschreibung hierzu an. Des Weiteren finden Sie in der Datei *generic-store.ts* eine generische Implementierung des Redux-Stores sowie in der Datei *services/stores/stores.ts* ein Beispiel für dessen Verwendung.

Die Subject.asObservable-Methode: nur den Observable-Teil eines Subjects zur Verfügung stellen

Die letzte, bislang noch nicht besprochenene Methode des `TaskStore` ist die `selectItems`-Methode:

```
private items$ = new BehaviorSubject<Task[]>([]);

selectItems(): Observable<Task[]> {
  return this.items$.asObservable();
}
```

Listing 12.32 »task-store.ts«: Verwendung der »asObservable«-Methode

Hier sehen Sie eine weitere interessante Funktion von *RxJS*-Subjects in Aktion: Mithilfe der `asObservable`-Methode erzeugen Sie aus dem zuvor erstellten `BehaviorSubject` ein neues Observable, das anschließend alle Werte weiterleitet, die auf dem `items$`-Stream geschrieben werden. Gegebenenfalls werden Sie sich an dieser Stelle fragen, wieso Sie hier nicht einfach direkt das `items$`-Subject an den Aufrufer der Funktion zurückliefern, da das Subject ja ohnehin gleichzeitig Observable und Observer ist. Auch wenn dies technisch gesehen stimmt, hätte diese Lösung in Bezug auf die Kapselung der Daten eine Schwachstelle: Durch die direkte Rückgabe des Subjects hätte der Aufrufer von `selectItems` anschließend die Möglichkeit, selbst neue Werte auf den Stream zu schicken. Diese Lücke wird durch die Verwendung der `asObservable`-Methode geschlossen!

12.3.4 Anbindung der TaskListComponent an den Store

Sie kennen nun alle Implementierungsdetails des reaktiven Daten-Service. Wirklich interessant wird die neue Architektur aber natürlich erst im Zusammenspiel mit den Komponenten der Anwendung.

Schauen Sie sich also zunächst die Anbindung des Stores in der Komponente an. Ich habe mich in diesem Zusammenhang dafür entschieden, alle Zugriffe auf den Store über den `TaskService` zu kapseln – die Komponenten besitzen somit einen zentralen Zugriffspunkt für alle Task-bezogenen Operationen. Innerhalb des `TaskService` wird dafür die `selectTasks`-Methode bereitgestellt:

```
export class TaskService {
  tasks$:Observable<Task[]>;
  constructor(private http: Http, private taskStore: TaskStore) {
  }
  selectTasks() {
    return this.taskStore.selectItems();
  }
  ...
}
```

Listing 12.33 »task.service.ts«: Veröffentlichung des »tasks$«-Observables

Innerhalb der `TaskListComponent` können Sie sich nun mit diesem Observable verbinden:

```
export class TaskListComponent {
  tasks$: Observable<Task[]>;
  ...
  ngOnInit() {
    this.tasks$ = this.taskService.selectTasks();
    ...
    merge(paramsStream$, searchTermStream$).pipe(
    distinctUntilChanged(),
    switchMap(query =>  this.taskService.findTasks(query)))
    .subscribe();
  }
}
```

Listing 12.34 »task-list.component.ts«: Verbindung mit dem Observable des TaskStores

Wie Sie sehen, wird die `tasks$`-Variable nun nur noch *ein einziges Mal* mit dem `Observable` des Stores verbunden. Der Typeahead-Mechanismus sorgt lediglich noch dafür, dass im `TaskService` die `findTasks`-Methode getriggert wird. Diese löst anschließend die `LOAD`-Action aus und sorgt dafür, dass der Store die angeforderten Werte in das `tasks$`-Observable pusht.

Der elementare Unterschied besteht also im Endeffekt darin, dass es der `TaskListComponent` nun »egal« ist, auf welchem Weg die Daten in den Store gelangen. Wird aus der Sidebar heraus ein Speichern ausgelöst, so gelangen diese Daten automatisch, ohne weiteres Zutun der Komponente, in die Liste. Schauen Sie sich, um den Unterschied zu verstehen, noch einmal die ursprüngliche Version vor der Umstellung an:

```
ngOnInit() {
  this.tasks$ = merge(paramsStream$, searchTermStream$).pipe(
    distinctUntilChanged(),
```



```
    switchMap(query =>  this.taskService.findTasks(query)));
  ...
  this.taskService.taskSaved$.subscribe((changedTask) => {
    this.tasks$ = this.taskService.findTasks(this.searchTerm.value);
  })
}
```

Listing 12.35 »task-list.component.ts«: ursprüngliche Version der Service-Anbindung

Sie sehen: Alle Logik zur Aktualisierung des States ist nun sauber im Store gekapselt. Die Komponente kann sich darauf verlassen, immer mit den korrekten Daten versorgt zu werden.

12.3.5 Der »In Bearbeitung«-Zähler

Auf Basis der neuen Architektur ist es nun auch ein Kinderspiel, den in Abschnitt 12.3.1 angesprochenen »In Bearbeitung«-Zähler im Header der Anwendung umzusetzen. Zur Integration des Zählers können Sie sich in der Klasse `AppComponent` einfach ebenfalls beim Store anmelden:

```
export class AppComponent {
  numberInProgress$: Observable<number>;
  ...
  ngOnInit() {
    this.numberInProgress$ = this.taskService.selectTasks().pipe(
      map(tasks => tasks.filter(t => t.state === 'IN_PROGRESS').length)
    );
  }
}
```

Listing 12.36 »app.component.ts«: Implementierung des »In Bearbeitung«-Zählers

Die Komponente wandelt den `tasks$`-Stream mithilfe des `map`-Operators in einen Stream um, der immer die aktuelle Anzahl an Tasks im Status `IN_PROGRESS` enthält. Mit insgesamt drei Zeilen Quellcode haben Sie nun also einen sich automatisch aktualisierenden Zähler implementiert. Innerhalb der Navigationsleiste können Sie den aktuellen Wert anschließend mithilfe der `AsyncPipe` ausgeben.

Bei der Verwendung des *Bootstrap*-CSS-Frameworks bietet sich hierfür die `badge`-Klasse an:

```
<span class="badge">
  In Bearbeitung: {{numberInProgress$ | async}}
</span>
```

Listing 12.37 »app.component.html«: Darstellung des Zählers

Ändern Sie nun eine Aufgabe über die Sidebar oder die Edit-Seite, wird der Zähler, wie in Abbildung 12.14 dargestellt, automatisch aktualisiert.

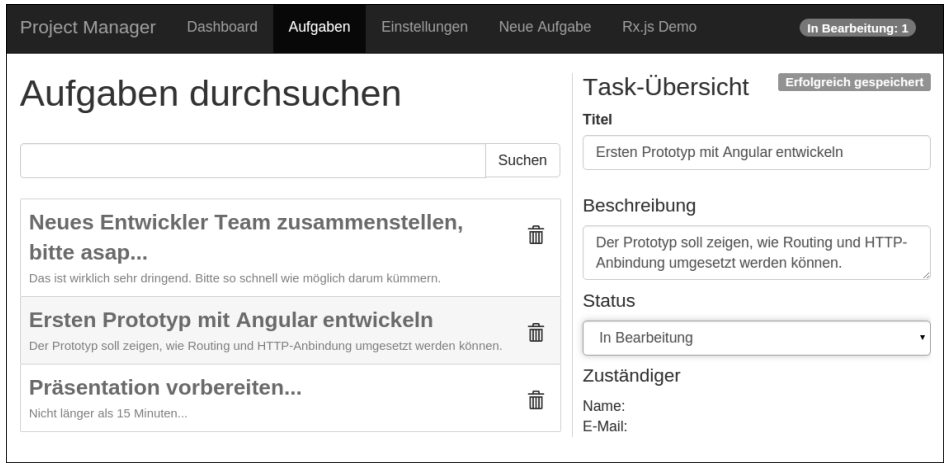

Abbildung 12.14 Darstellung des »In Bearbeitung«-Zählers

Die Bibliotheken ngrx und ngxs

Sie kennen nun die grundlegenden Techniken für die Arbeit mit einem reaktiven Data-Store. Insbesondere für das Verständnis der zugrunde liegenden Konzepte ist es an dieser Stelle absolut sinnvoll, den reaktiven Workflow einmal von Grund auf zu implementieren. Für den Einsatz in Ihrem eigenen Projekt möchte ich Ihnen hier aber dennoch die Verwendung einer bestehenden Store-Implementierung wie *ngrx* (*https://github.com/ngrx/store*) oder *ngxs* (*https://github.com/ngxs/store*) ans Herz legen. Beide Bibliotheken basieren auf den in diesem Abschnitt vorgestellten Grundideen, bieten Ihnen zusätzlich aber einige weitere Annehmlichkeiten, wie z. B. Lazy-Loading-Unterstützung oder einen Live-Viewer für den aktuellen Inhalt des Stores.

12.4 Anbindung von Websockets zur Implementierung einer Echtzeitanwendung

Sollten Sie bislang immer noch nicht von der neuen reaktiven Datenarchitektur überzeugt sein, hoffe ich, dass ich Sie spätestens mit diesem Abschnitt von der Mächtigkeit des Konzepts überzeugen kann. Auf den folgenden Seiten möchte ich Ihnen zeigen, wie Sie Ihre Anwendung mit wenigen Zeilen Code in eine echtzeitfähige Webapplikation verwandeln können. Das Ziel besteht darin, dass sich der Zustand der Anwendung über verschiedene Browserfenster hinweg synchronisiert: Arbeiten mehrere Personen an der Abarbeitung und Erstellung der Tasks, so sollen neu angelegte Tasks ohne einen Reload in allen Browserfenstern erscheinen.

Zur Umsetzung dieses Features wird die Bibliothek *Socket.IO* verwendet. Hierbei handelt es sich um eine sehr weit verbreitete Bibliothek zur Implementierung von Websocket-Funktionalität.

12.4.1 Der Websocket-Server

Der im *project-manager-reactive*-Projekt mitgelieferte *projects-server* besitzt hierfür bereits eine sehr einfache Umsetzung der Websocket-Logik. Listing 12.38 zeigt den entsprechenden Ausschnitt aus der Datei */projects-server/server.js:*

```
const io = require('socket.io')(3001);
const _socketMap = {};
io.on('connection', (socket) => {
  _socketMap[socket.id] = socket;
  socket.on('broadcast_task', (data) => {
    for (const socketKey in _socketMap) {
      const broadcastTo = _socketMap[socketKey];
      if (socket.id !== broadcastTo.id) {
        broadcastTo.emit('task_saved', data)
      }
    }
  });
});
```

Listing 12.38 »server.js«: simple Umsetzung der Websocket-Funktionalität im Server

Ich möchte an dieser Stelle nicht zu sehr in die Details der Websocket-Entwicklung einsteigen. Zusammenfassend gesagt, sorgt der obige Ausschnitt aber dafür, dass sich Browser über den Port `3001` mit dem Websocket verbinden können. Wird eine neue Verbindung erstellt, so wird diese in der `_socketMap` gespeichert. Empfängt der Server eine Nachricht vom Typ `broadcast_task`, so wird dieser Task an alle angemeldeten Websocket-Verbindungen weitergeleitet (außer an diejenige, die den Broadcast ausgelöst hat). Die Weiterleitung des `Task` erfolgt dabei ebenfalls über eine Websocket-Nachricht mit dem Typ `task_saved`.

Abbildung 12.15 zeigt den Ablauf, der implementiert werden soll, um die Anwendung echtzeitfähig zu machen.

Hinweis zur Server-Implementierung

In einer »echten« Backend-Applikation würde der Broadcast vermutlich durch den Endpunkt ausgelöst werden, der für das Speichern des Tasks zuständig ist. Da diese Implementierung in Verbindung mit dem verwendeten *json-server* etwas aufwendiger ist, habe ich mich an dieser Stelle entschieden, die Verantwortung für den Broadcast in den Client zu verlagern.

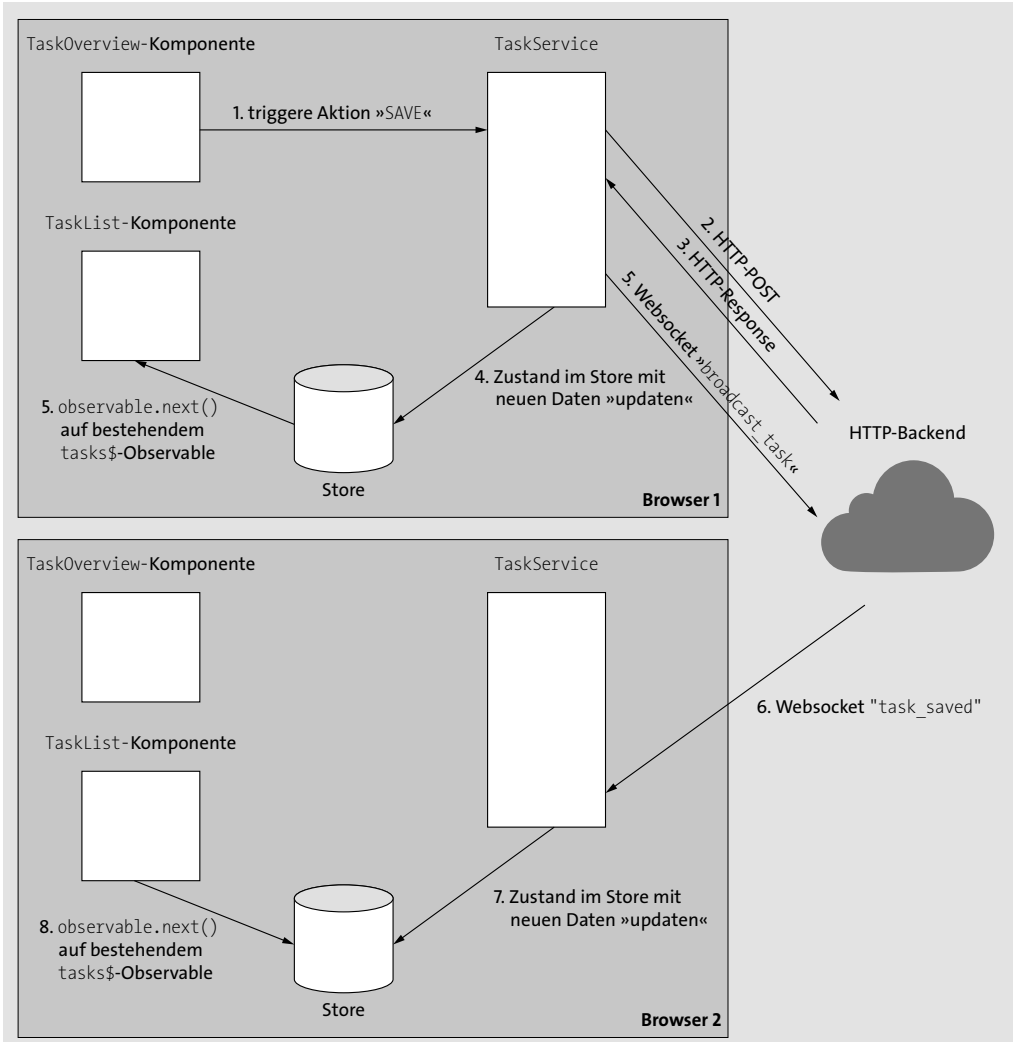

Abbildung 12.15 Ablauf bei der Arbeit mit mehreren Browserfenstern

Zusätzlich zum bisherigen Ablauf wird nach dem Speichern des Tasks der Websocket-Aufruf `broadcast_task` ausgelöst. Der Server schickt daraufhin die `task_saved`-Message an alle weiteren Browser. Wie Sie sehen, wird die Nachricht im zweiten Browser automatisch entgegengenommen. Die Komponente wird anschließend ohne eigenes Zutun über die Änderungen informiert und kann sich aktualisieren. Die Besonderheit besteht somit erneut darin, dass es für die Komponente völlig unerheblich ist, wie der Service seine Daten erhält – sie bleibt von der Websocket-Anbindung also absolut unberührt.

Um die Websocket-Funktionalität zu implementieren, müssen Sie im Wesentlichen zwei Dinge tun:

1. beim Speichern die Websocket-Nachricht auslösen
2. im `TaskService` auf eingehende Websocket-Nachrichten reagieren

12.4.2 Integration von Socket.IO in die Anwendung

Um *Socket.IO* in Ihrer Anwendung zu verwenden, ist es zunächst notwendig, dass Sie die Schritte zur Verwendung von Drittanbieter-Bibliotheken durchführen, die ich bereits in Kapitel 2, »Das Angular-CLI: professionelle Projektorganisation für Angular-Projekte«, vorgestellt habe. Im Fall von Socket.IO erfolgt dies über folgenden Befehl:

```
npm install socket.io-client
```

Bei der Integration von Socket.IO in die Anwendung bietet es sich nun an, die Funktion zur Erzeugung von Websocket-Verbindungen (`io`) nicht direkt in den `TaskService` zu importieren, sondern über einen Provider in der Applikation bekannt zu machen. Auch wenn dies nicht zwingend erforderlich ist, ermöglicht diese Technik es Ihnen, die Funktion im Test leicht auszutauschen (siehe Kapitel 13, »Komponenten- und Unit-Tests: das Angular-Testing-Framework«). Wie in Kapitel 7, »Services und Dependency-Injection: lose Kopplung für Ihre Business-Logik«, beschrieben, bietet sich für solche Anwendungsfälle die Einführung eines `InjectionToken` an:

```
import {InjectionToken} from '@angular/core';
...
export const SOCKET_IO = new InjectionToken<any>('socket-io');
```

Listing 12.39 »app.tokens.ts«: die »InjectionToken«-Instanz für Socket.IO erstellen

Erweitern Sie anschließend Ihr Hauptmodul wie folgt:

```
import {io, Socket } from 'socket.io-client';
import {SOCKET_IO} from './app.tokens';
...
export function socketIoFactory(): (url: string) => Socket {
  return io;
}
@NgModule({
  providers: [
     ...
    {provide: SOCKET_IO, useFactory: socketIoFactory},
  ],
  bootstrap: [AppComponent]
```

```
})
export class AppModule {
}
```

Listing 12.40 »app.module.ts«: Bereitstellung der Funktion »io« über ein Token

Die von Socket.IO bereitgestellte Funktion `io` wird jetzt über das InjectionToken `SOCKET_IO` der Anwendung zur Verfügung gestellt.

12.4.3 Verwendung von Socket.IO im TaskService

Innerhalb des `TaskService` können Sie die Funktion nun über den Konstruktor injizieren. Der Aufbau der Websocket-Verbindung erfolgt anschließend über den Aufruf der Funktion mit der Websocket-URL:

```
import {SOCKET_IO} from '../../app.tokens';
const WEB_SOCKET_URL = 'http://localhost:3001';
...
export class TaskService {
  socket: Socket;
  constructor(private http:HttpClient, private taskStore:TaskStore,
              @Inject(SOCKET_IO) socketIO: (url: string) => Socket) {
    this.socket = socketIO(WEB_SOCKET_URL);
  ...
  }
}
```

Listing 12.41 »task.service.ts«: Injektion der »io«-Funktion und Aufbau der Websocket-Verbindung

Beachten Sie hier auch die Verwendung des TypeScript-Interface `Socket`. So können Sie über dieses Interface auch bei der Arbeit mit *Socket.IO* von Typsicherheit und Auto-Completion profitieren.

Der nächste Schritt besteht nun darin, beim Speichern eines Tasks andere Nutzer über die ausgeführte Änderung zu informieren:

```
export class TaskService {
...
  saveTask(task: Task): Observable<Task> {
    return this.http.request<Task>(method, `${BASE_URL}/${task.id ?? ''}`,{
         body: task
       }).pipe(
      tap(savedTask => {
        const actionType = task.id ? EDIT :ADD;
```



```
        const action = {type: actionType, data: savedTask};
        this.taskStore.dispatch(action);
        this.socket.emit('broadcast_task', action);
      }));
  }
}
```

Listing 12.42 »task.service.ts«: die auszuführende Action per Websocket versenden

Anstatt die Action, die für die Statusveränderung zuständig ist, nur an den eigenen Store zu übergeben, wird sie hier zusätzlich mithilfe der `emit`-Methode per Websocket an den Server übertragen. Da es sich bei Redux-Actions um ganz normale JavaScript-Objekte handelt, ist es problemlos möglich, diese als JSON-Objekte zu versenden.

Jetzt fehlt nur noch die Auswertung von eingehenden Websocket-Nachrichten. Listing 12.43 zeigt die hierfür notwendigen Erweiterungen am `TaskService`:

```
import {..., fromEvent} from 'rxjs';
...
export class TaskService {
  ...
  constructor(private http: HttpClient, private taskStore: TaskStore,
              @Inject(SOCKET_IO) socketIO: (url: string) => Socket) {
    this.socket = socketIO(WEB_SOCKET_URL);
    fromEvent<Action>(this.socket, 'task_saved')
      .subscribe((action) => {
        this.taskStore.dispatch(action);
      });
  }
  ...
}
```

Listing 12.43 »task.service.ts«: Verarbeitung der eingehenden Websocket-Nachricht

Mithilfe des Erzeugungsoperators `fromEvent` lauschen Sie auf eingehende Nachrichten. Dabei können Sie über einen Generic-Typ mitgeben, welchen Typ (in diesem Fall `Action`) Sie erwarten. Kommt eine entsprechende Nachricht an, leiten Sie diese einfach an den `taskStore` weiter.

Fertig! Mit einigen wenigen Zeilen Quellcode haben Sie Ihre Anwendung echtzeitfähig gemacht. Öffnen Sie zur Überprüfung der Funktionalität die Anwendung in zwei getrennten Browserfenstern, und öffnen Sie in einem Fenster die Task-Liste. Im anderen Fenster können Sie je nach Belieben einen neuen Task über das Formular anlegen oder einen Task über die Sidebar editieren. Alle Änderungen werden sich

nun automatisch im anderen Browserfenster widerspiegeln – und als i-Tüpfelchen verändert sich, wie Sie in Abbildung 12.16 sehen können, zusätzlich der »In Bearbeitung«-Zähler.

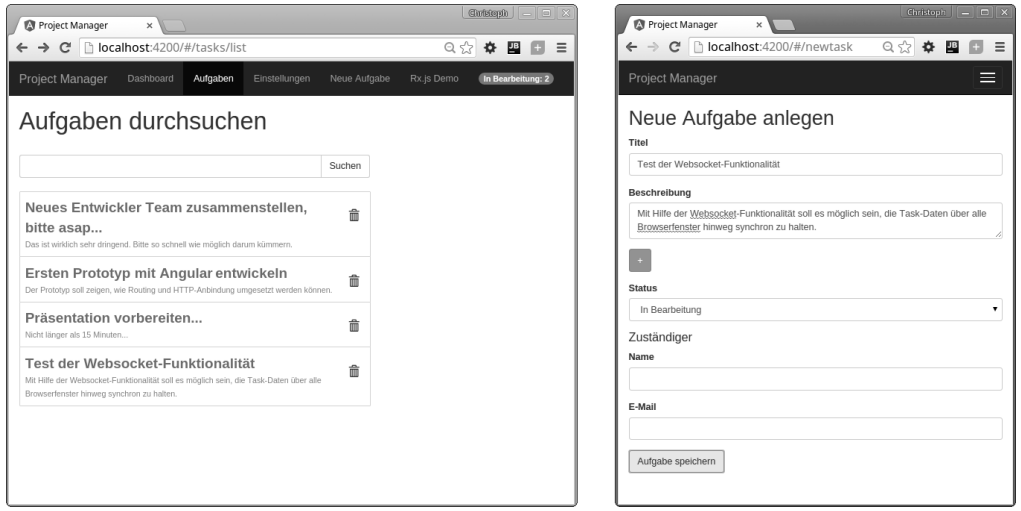

Abbildung 12.16 Synchronisation von zwei Browserfenstern über Websockets

12.5 ChangeDetectionStrategy.OnPush: Performance-Schub durch die reaktive Architektur

Neben den architektonischen Verbesserungen für Ihre Applikation bietet der vorgestellte Ansatz Ihnen noch einen weiteren interessanten Vorteil. So besteht einer der wichtigsten Faktoren für die Performance Ihrer Anwendung in der Zahl der Change-Detection-Aufrufe. Details zu diesem Thema haben Sie bereits in Kapitel 5, »Fortgeschrittene Komponenten-Konzepte«, kennengelernt.

Wie ich dort bereits sagte, sollten Sie nach Möglichkeit versuchen, aufwendige Komponenten (insbesondere Listen) mithilfe der ChangeDetection-Strategie `OnPush` so zu konfigurieren, dass diese sich nur dann neu zeichnen, wenn sich Input-Bindings geändert haben. Durch die Verwendung des Redux-ähnlichen Stores und die Erzeugung von *Immutable Data Structures* haben Sie hierfür nun schon alle Voraussetzungen geschaffen: Beim Speichern eines Tasks werden automatisch ein neues Task- und ein neues Task-Listen-Objekt erzeugt. Sie können nun also bedenkenlos die `TaskItemComponent` und die `TaskListComponent` mit der `OnPush`-Strategie versehen:

```
import {ChangeDetectionStrategy} from '@angular/core';
@Component({
  changeDetection: ChangeDetectionStrategy.OnPush,
  ...
})
export class TaskListComponent {
  ...
}
```

Listing 12.44 »task-list.component.ts«: Verwendung der »OnPush«-Strategie

```
@Component({
  changeDetection: ChangeDetectionStrategy.OnPush,
  ...
})
export class TaskItemComponent {
  ...
  ngAfterViewChecked() {
    const taskId = (this.task ? this.task.id : '');
    console.log(`Task ${taskId} checked ${++this.checkCnt} times`)
  }
}
```

Listing 12.45 »task-item.component.ts«: Verwendung der »OnPush«-Strategie und Logging jedes Komponenten-Checks

Starten Sie die Anwendung nun erneut, werden Sie feststellen, dass der Change-Detection-Mechanismus die Komponente nur noch dann überprüft, wenn tatsächlich Änderungen am relevanten Modell stattgefunden haben!

12.6 Zusammenfassung und Ausblick

Ich hoffe, ich konnte Sie mit diesem Kapitel von den Vorzügen der reaktiven Programmierung überzeugen! So bietet der konsequente Einsatz von RxJS in sämtlichen Teilbereichen von Angular Ihnen beste Voraussetzungen für die Entwicklung von Event-getriebenen Anwendungen. Bislang haben Sie die Themen *Formulare*, *Routing* und *HTTP* weitestgehend isoliert betrachtet. In diesem Kapitel haben Sie aber gesehen, wie die einzelnen Bestandteile durch den geschickten Einsatz von RxJS-Operatoren sehr elegant miteinander verbunden werden können.

Die folgende Liste gibt Ihnen noch einmal einen Überblick über die wichtigsten Erkenntnisse dieses Kapitels:

- RxJS stellt Ihnen eine Vielzahl an vordefinierten `Observable`-Typen und Operatoren für die Umsetzung von reaktiven Anwendungen zur Verfügung.
- Angular setzt bei der Umsetzung von asynchroner Funktionalität modulübergreifend auf RxJS. So stellen unter anderem das Routing-, das Formular- und das HTTP-Modul Observables für die Auswertung von asynchronen Vorgängen bereit.

- Die Operatoren `merge`, `mergeMap` und `switchMap` spielen eine zentrale Rolle für die Kombination mehrerer Observables.
- In einer Push-getriebenen Datenarchitektur werden Komponenten automatisch über neue Daten informiert. (Die Daten werden in die Komponente »gepusht«.)
- *Redux* ist eine Technik, bei der der Applikationszustand zu jedem Zeitpunkt in einem zentralen Objekt im Client zur Verfügung steht.
- Die Kombination des Redux-Ansatzes mit RxJS-Observables bietet Ihnen die Möglichkeit, alle Änderungen Ihres Applikationszustands an einer zentralen Stelle zu verwalten und Ihre Komponenten durch einen Push-Mechanismus über Änderungen zu informieren.
- Die bei Redux verwendeten *Immutable Data Structures* ermöglichen Ihnen automatisch die Verwendung der `ChangeDetection.OnPush`-Strategie, um effizient auf Datenänderungen reagieren zu können.
- Die Anmeldung an einem `Observable` erfolgt über den Aufruf der `subscribe`-Methode. Die Methode liefert ein Objekt der Klasse `Subscription` zurück.
- Die spätere Abmeldung erfolgt durch den Aufruf der `unsubscribe`-Methode auf dieser `Subscription`.
- Eine Alternative zur expliziten Abmeldung der `Subscription` besteht in der Verwendung des `takeUntil`-Operators.
- RxJS-Subjects stellen Ihnen eine elegante Möglichkeit zur Implementierung von Multicasting-Funktionalität zur Verfügung.
- Neben den von Angular bereitgestellten Observables bietet RxJS Ihnen eine Vielzahl an Möglichkeiten, vordefinierte Observable-Arten zu instanziieren.

Die Project-Manager-Anwendung kombiniert nun bereits diverse Bestandteile des Angular-Frameworks, um Ihren Nutzern eine moderne, bedienerfreundliche Webanwendung zur Verfügung zu stellen. Nun ist es an der Zeit, dafür zu sorgen, dass dies auch dauerhaft so bleibt: Im folgenden Kapitel werden Sie lernen, wie Sie mithilfe des Angular-Testing-Frameworks umfangreiche Komponenten-Tests für Ihre Anwendung implementieren können. Das darauffolgende Kapitel wird sich mit Integrationstests auf Basis von Cypress beschäftigen.