Ferdinand Malcher · Johannes Hoppe · Danny Koppenhagen

Angular

Grundlagen, fortgeschrittene Themen
und Best Practices

inklusive
NativeScript
und NgRx

iX EDITION

dpunkt.verlag

Leseprobe

Angular, 2. Auflage

https://angular-buch.com

Liebe Leserin, lieber Leser,

das Angular-Ökosystem wird kontinuierlich verbessert. Bitte haben Sie Verständnis dafür, dass sich beim Entwickeln einige Dinge vom gedruckten Stand unterscheiden können. Die Github-Repositorys halten wir hierzu stets auf dem neuesten Stand.

Unter *https://angular-buch.com/updates* informieren wir Sie ausführlich über Breaking Changes und neue Funktionen. Wir freuen uns auf Ihren Besuch.

Sollten Sie einen Fehler vermuten oder einen Breaking Change entdeckt haben, so bitten wir Sie um Ihre Mithilfe! Bitte kontaktieren Sie uns hierfür unter *team@angular-buch.com* mit einer Beschreibung des Problems. Wir werden eine Lösung auf unserer Website bekannt machen, sodass alle Leser des Buchs davon profitieren können.

Wir wünschen Ihnen viel Spaß mit Angular!

Alles Gute
Ferdinand, Johannes, Danny

Ferdinand Malcher arbeitet als selbstständiger Entwickler, Berater und Mediengestalter mit Schwerpunkt auf Angular, TypeScript und Node.js. Gemeinsam mit Johannes Hoppe hat er die Angular.Schule gegründet und bietet Workshops und Beratung zu Angular an.

@fmalcher01

Johannes Hoppe arbeitet als selbständiger Softwarearchitekt und Berater für Angular, .NET und Node.js. Zusammen mit Ferdinand Malcher hat er die Angular.Schule gegründet. Für seine Community-Tätigkeit rund ums Web wurde er mehrfach als Progress Developer Expert ausgezeichnet. Johannes ist Organisator des Angular Heidelberg Meetup.

@JohannesHoppe

Danny Koppenhagen arbeitet als Berater und Entwickler von Client- und Serveranwendungen mit NodeJS, TypeScript und Angular. Sein Schwerpunkt liegt in der Entwicklung von Webanwendungen im Enterprise-Umfeld sowie auf Apps mit NativeScript.

@d_koppenhagen

Sie erreichen das Autorenteam auf Twitter unter *@angular_buch*.

Ferdinand Malcher · Johannes Hoppe · Danny Koppenhagen

Angular

Grundlagen, fortgeschrittene Themen und Best Practices – inklusive NativeScript und NgRx

2., aktualisierte und erweiterte Auflage

dpunkt.verlag

iX EDITION

***iX*-Edition**
In der *iX*-Edition erscheinen Titel, die vom dpunkt.verlag gemeinsam mit der Redaktion der Computerzeitschrift iX ausgewählt und konzipiert werden. Inhaltlicher Schwerpunkt dieser Reihe sind Software- und Webentwicklung sowie Administration.

Ferdinand Malcher, Johannes Hoppe, Danny Koppenhagen
team@angular-buch.com

Lektorat: René Schönfeldt
Lektoratsassistenz/Projektkoordinierung: Anja Weimer
Copy-Editing: Anette Schwarz, Ditzingen
Satz: Da-TeX, Leipzig
Herstellung: Stefanie Weidner
Umschlaggestaltung: Helmut Kraus, *www.exclam.de*
Druck und Bindung: mediaprint solutions GmbH, 33100 Paderborn

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.d-nb.de* abrufbar.

ISBN:
Print 978-3-86490-646-6
PDF 978-3-96088-712-6
ePub 978-3-96088-713-3
mobi 978-3-96088-714-0

2., aktualisierte und erweiterte Auflage 2019

Wieblinger Weg 17
69123 Heidelberg

Hinweis:
Dieses Buch wurde auf PEFC-zertifiziertem Papier aus nachhaltiger Waldwirtschaft gedruckt. Der Umwelt zuliebe verzichten wir zusätzlich auf die Einschweißfolie.

Schreiben Sie uns:
Falls Sie Anregungen, Wünsche und Kommentare haben, lassen Sie es uns wissen: hallo@dpunkt.de.

5 4 3 2 1 0

Vorwort

»Angular is one of the most adopted frameworks on the planet.«

Brad Green
(Angular Engineering Director)

Angular ist eines der populärsten Frameworks für die Entwicklung von Single-Page-Applikationen. Das Framework wird weltweit von großen Unternehmen eingesetzt, um modulare, skalierbare und gut wartbare Applikationen zu entwickeln. Tatsächlich hat Angular seinen Ursprung beim wohl größten Player des Internets – Google. Obwohl kommerzielle Absichten hinter der Idee stehen, wurde Angular von Anfang an quelloffen unter der MIT-Lizenz veröffentlicht. Im September 2016 erschien Angular in der Version 2.0.0. Google setzte damit einen Meilenstein in der Welt der modernen Webentwicklung: Das Framework nutzt die Programmiersprache TypeScript, bietet ein ausgereiftes Tooling und komponentenbasierte Entwicklung. In kurzer Zeit haben sich rund um Angular ein umfangreiches Ökosystem und eine vielfältige Community gebildet.

Die Entwicklung wird maßgeblich von einem dedizierten Team bei Google vorangetrieben, wird aber auch stark aus der Community beeinflusst. Angular gilt neben React.js (Facebook) und Vue.js (Community-Projekt) als eines der weltweit beliebtesten Webframeworks. Sie haben also die richtige Entscheidung getroffen und haben Angular für die Entwicklung Ihrer Projekte ins Auge gefasst.

Das Framework ist modular aufgebaut und stellt eine Vielzahl an Funktionalitäten bereit, um wiederkehrende Standardaufgaben zu lösen. Der Einstieg ist umfangreich, aber die Konzepte sind durchdacht und konsequent. Hat man die Grundlagen erlernt, so kann man den Fokus auf die eigentliche Businesslogik legen. Häufig verwendet man im Zusammenhang mit Angular das Attribut *opinionated*, das wir im Deutschen mit dem Begriff *meinungsstark* ausdrücken können: Angular ist ein meinungsstarkes Framework, das viele klare Richtlinien zu Architektur, Codestruktur und Best Practices definiert. Das kann zu Anfang umfangreich erscheinen, sorgt aber dafür, dass in der gesam-

Opinionated Framework

ten Community einheitliche Konventionen herrschen, Standardlösungen existieren und bestehende Bibliotheken vorausgewählt wurden.

Obwohl die hauptsächliche Zielplattform für Angular-Anwendungen der Browser ist, ist das Framework nicht darauf festgelegt: Durch seine Plattformunabhängigkeit kann Angular auf nahezu jeder Plattform ausgeführt werden, unter anderem auf dem Server und nativ auf Mobilgeräten.

Grundlegende Konzepte

Sie werden in diesem Buch lernen, wie Sie mit Angular komponentenbasierte Single-Page-Applikationen entwickeln. Wir werden Ihnen vermitteln, wie Sie Abhängigkeiten und Asynchronität mithilfe des Frameworks behandeln. Weiterhin erfahren Sie, wie Sie mit Routing die Navigation zwischen verschiedenen Teilen der Anwendung implementieren. Sie werden lernen, wie Sie komplexe Formulare mit Validierungen in Ihre Anwendung integrieren und wie Sie Daten aus einer HTTP-Schnittstelle konsumieren können.

Beispielanwendung

Wir entwickeln mit Ihnen gemeinsam eine Anwendung, anhand derer wir Ihnen all diese Konzepte von Angular beibringen. Dabei führen wir Sie Schritt für Schritt durch das Projekt – vom Projektsetup über das Testen des Anwendungscodes bis zum Deployment der fertig entwickelten Anwendung. Auf dem Weg stellen wir Ihnen eine Reihe von Tools, Tipps und Best Practices vor, die wir in mehr als zwei Jahren Praxisalltag mit Angular sammeln konnten.

Nach dem Lesen des Buchs sind Sie in der Lage,

- das Zusammenspiel der Funktionen von Angular sowie das Konzept hinter dem Framework zu verstehen,
- modulare, strukturierte und wartbare Webanwendungen mithilfe des Angular-Frameworks zu entwickeln sowie
- durch die Entwicklung von Tests qualitativ hochwertige Anwendungen zu erstellen.

Die Entwicklung von Angular macht vor allem eines: Spaß! Diesen Enthusiasmus für das Framework und für Webtechnologien möchten wir Ihnen in diesem Buch vermitteln – wir nehmen Sie mit auf die Reise in die Welt der modernen Webentwicklung!

Versionen und Namenskonvention: Angular vs. AngularJS

In diesem Buch dreht sich alles um das Framework Angular. Sucht man nach dem Begriff »Angular« im Internet, so stößt man auch oft noch auf die Bezeichnung »AngularJS«. Hinter dieser Bezeichnung verbirgt sich die Version 1 des Frameworks. Mit der Version 2 wurde Angular von Grund auf neu entwickelt. Die offizielle Bezeichnung für das neue Framework ist *Angular*, ohne Angabe der Programmiersprache und ohne eine spezifische Versionsnummer. Angular erschien im September 2016 in der Version 2.0.0 und hat viele neue Konzepte und Ideen in die Community gebracht. Weil es sich um eine vollständige Neuentwicklung handelt, ist Angular nicht ohne Weiteres mit dem alten AngularJS kompatibel. Um Verwechslungen auszuschließen, gilt also die folgende Konvention:

It's just »Angular«.

- **Angular** – das Angular-Framework ab **Version 2 und höher** (dieses Buch ist durchgängig auf dem Stand von Angular 7 bzw. der Beta-Version von Angular 8)
- **AngularJS** – das Angular-Framework in der **Version 1.x.x**

AngularJS, das 2009 erschien, ist zwar mittlerweile etwas in die Jahre gekommen, viele Webanwendungen setzen aber weiterhin auf das Framework. Die letzte Version 1.7 wurde im Sommer 2018 veröffentlicht und wird nach einer dreijährigen Phase mit Long Term Support (LTS) ab Juli 2021 nicht mehr unterstützt.[1]

Long Term Support für AngularJS

Sie haben also die richtige Entscheidung getroffen, Angular ab Version 2.0.0 einzusetzen. Diese Versionsnummer *x.y.z* basiert auf *Semantic Versioning*.[2] Der Release-Zyklus von Angular ist kontinuierlich geplant: Im Rhythmus von sechs Monaten erscheint eine neue Major-Version *x*. Die Minor-Versionen *y* werden monatlich herausgegeben, nachdem eine Major-Version erschienen ist.

Semantic Versioning

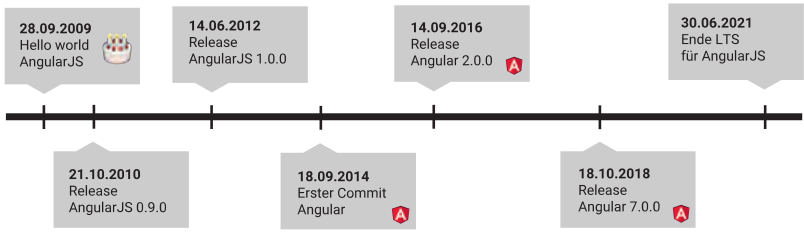


Abb. 1
Zeitleiste der Entwicklung von Angular

[1] https://ng-buch.de/a/1 – AngularJS: Version Support Status

[2] https://ng-buch.de/a/2 – Semantic Versioning 2.0.0

Umgang mit Aktualisierungen

Das Release einer neuen Major-Version von Angular bedeutet keineswegs, dass alle Ideen verworfen werden und Ihre Software nach einem Update nicht mehr funktioniert. Auch wenn Sie eine neuere Angular-Version verwenden, behalten die in diesem Buch beschriebenen Konzepte ihre Gültigkeit. Die Grundideen von Angular sind seit Version 2 konsistent und auf Beständigkeit über einen langen Zeitraum ausgelegt. Alle Updates zwischen den Major-Versionen waren in der Vergangenheit problemlos möglich, ohne dass Breaking Changes die gesamte Anwendung unbenutzbar machen. Gibt es doch gravierende Änderungen, so werden stets ausführliche Informationen und Tools zur Migration angeboten.

Alle Beispiele aus diesem Buch sowie zusätzliche Links und Hinweise können Sie über eine zentrale Seite erreichen:

Die Begleitwebsite zum Buch

https://angular-buch.com

Unter anderem veröffentlichen wir dort zu jeder Major-Version einen Artikel mit den wichtigsten Neuerungen und den nötigen Änderungen am Beispielprojekt. Wir empfehlen Ihnen aus diesem Grund, unbedingt einen Blick auf die Begleitwebsite des Buchs zu werfen, bevor Sie beginnen, sich mit den Inhalten des Buchs zu beschäftigen.

An wen richtet sich das Buch?

Webentwickler mit JavaScript-Erfahrung

Dieses Buch richtet sich an Webentwickler, die einige Grundkenntnisse mitbringen. Wir setzen allgemeine Kenntnisse in JavaScript voraus. Wenn Sie bereits ein erstes JavaScript-Projekt umgesetzt haben und Ihnen Frameworks wie jQuery vertraut sind, werden Sie an diesem Buch sehr viel Freude haben. Mit Angular erwartet Sie das modulare Entwickeln von Single-Page-Applikationen in Kombination mit Unit- und UI-Testing.

TypeScript-Einsteiger und Erfahrene

Für die Entwicklung mit Angular nutzen wir die populäre Programmiersprache TypeScript. Doch keine Angst: TypeScript ist lediglich eine Erweiterung von JavaScript, und die neuen Konzepte sind sehr eingängig und schnell gelernt.

Praxisorientierte Einsteiger

In diesem Buch wird ein praxisorientierter Ansatz verfolgt. Sie werden anhand einer Beispielanwendung schrittweise die Konzepte und Funktionen von Angular kennenlernen. Dabei lernen Sie nicht nur die Grundlagen kennen, sondern wir vermitteln Ihnen auch eine Vielzahl von Best Practices und Erkenntnissen aus mehrjähriger Praxis mit Angular.

Was sollten Sie mitbringen?

Grundkenntnisse in JavaScript, HTML und CSS

Da wir Erfahrungen in der Webentwicklung mit JavaScript voraussetzen, ist es für jeden Entwickler, der auf diesem Gebiet unerfahren ist, empfehlenswert, sich die nötigen Grundlagen zu erarbeiten. Darüber hinaus sollten Sie Grundkenntnisse im Umgang mit HTML und CSS mitbringen. Der *dpunkt.verlag* bietet eine große Auswahl an Einstiegsliteratur für HTML, JavaScript und CSS an. Sollten Sie über keinerlei TypeScript-Kenntnisse verfügen: kein Problem! Alles, was Sie über TypeScript wissen müssen, um die Inhalte dieses Buchs zu verstehen, wird in einem separaten Kapitel vermittelt.

Keine Angular-Vorkenntnisse nötig!

Sie benötigen *keinerlei* Vorkenntnisse im Umgang mit Angular bzw. AngularJS. Ebenso müssen Sie sich nicht vorab mit benötigten Tools und Hilfsmitteln für die Entwicklung von Angular-Applikationen vertraut machen. Das nötige Wissen darüber wird Ihnen in diesem Buch vermittelt.

Für wen ist dieses Buch weniger geeignet?

Unerfahrene Webentwickler

Um Inhalte des Buchs zu verstehen, werden Erfahrungen im Webumfeld vorausgesetzt. Entwicklern ohne Vorkenntnisse in diesem Umfeld wird der Einstieg schwerer fallen. Sie sollten sich zunächst die grundlegenden Kenntnisse in den Bereichen HTML, JavaScript und CSS aneignen.

Kein Nachschlagewerk

Weiterhin ist dieses Buch nicht als Nachschlagewerk zu verstehen. Es werden nicht alle Möglichkeiten und Konzepte von Angular bis in jedes Detail erläutert. Anhand des Beispielprojekts werden die wichtigsten Funktionalitäten praxisorientiert vermittelt. Um Details zu den einzelnen Framework-Funktionen nachzuschlagen, empfehlen wir die offizielle Dokumentation für Entwickler.[3]

Offizielle Angular-Dokumentation

[3] https://ng-buch.de/a/3 – Angular Docs

Wie ist dieses Buch zu lesen?

Einführung, Tools und Schnellstart

Wir beginnen im ersten Teil des Buchs mit einer Einführung, in der Sie alles über die verwendeten Tools und benötigtes Werkzeug erfahren. Im Schnellstart tauchen wir sofort in Angular ein und nehmen Sie mit zu einem schnellen Einstieg in das Framework und den Grundaufbau einer Anwendung.

Einführung in TypeScript

Der zweite Teil vermittelt Ihnen einen Einstieg in TypeScript. Sie werden hier mit den Grundlagen dieser typisierten Skriptsprache vertraut gemacht und erfahren, wie Sie die wichtigsten Features verwenden können. Entwickler, die bereits Erfahrung im Umgang mit TypeScript haben, können diesen Teil überspringen.

Beispielanwendung

Der dritte Teil ist der Hauptteil des Buchs. Hier möchten wir mit Ihnen zusammen eine Beispielanwendung entwickeln. Die Konzepte und Technologien von Angular wollen wir dabei direkt am Beispiel vermitteln. So stellen wir sicher, dass das Gelesene angewendet wird und jeder Abschnitt automatisch einen praktischen Bezug hat.

Iterationen

Nach einer Projekt- und Prozessvorstellung haben wir das Buch in mehrere Iterationen eingeteilt. In jeder Iteration gilt es Anforderungen zu erfüllen, die wir gemeinsam mit Ihnen implementieren.

- Iteration I: Komponenten & Template-Syntax (ab S. 69)
- Iteration II: Services & Routing (ab S. 125)
- Iteration III: HTTP & reaktive Programmierung (ab S. 181)
- Iteration IV: Formularverarbeitung & Validierung (ab S. 261)
- Iteration V: Pipes & Direktiven (ab S. 339)
- Iteration VI: Module & fortgeschrittenes Routing (ab S. 387)
- Iteration VII: Internationalisierung (ab S. 433)

Storys

Eine solche Iteration ist in mehrere Storys untergliedert, die jeweils ein Themengebiet abdecken. Eine Story besteht immer aus einer theoretischen Einführung und der praktischen Implementierung im Beispielprojekt. Neben Storys gibt es Refactoring-Abschnitte. Dabei handelt es sich um technische Anforderungen, die die Architektur oder den Codestil der Anwendung verbessern.

Refactoring

Powertipps

Haben wir eine Iteration abgeschlossen, prüfen wir, ob wir unseren Entwicklungsprozess vereinfachen und beschleunigen können. In den *Powertipps* demonstrieren wir hilfreiche Werkzeuge, die uns bei der Entwicklung zur Seite stehen.

Testing

Nachdem alle Iterationen erfolgreich absolviert wurden, wollen wir das Thema *Testing* genauer betrachten. Hier erfahren Sie, wie Sie Ihre Angular-Anwendung automatisiert testen und so die Softwarequalität sichern können. Dieses Kapitel kann sowohl nach der Entwicklung des Beispielprojekts als auch parallel dazu bestritten werden.

Deployment

Im vierten Teil dreht sich alles um das Deployment einer Angular-Anwendung. Sie werden erfahren, wie Sie eine fertig entwickelte Angular-Anwendung fit für den Produktiveinsatz machen.

Weiterführende Themen

Im fünften und letzten Teil möchten wir Ihnen mit Server-Side Rendering, der Redux-Architektur und dem Framework NativeScript drei Ansätze näherbringen, die über eine Standardanwendung hinausgehen.

SSR

Mit *Server-Side Rendering (SSR)* machen Sie Ihre Anwendung fit für Suchmaschinen und verbessern zusätzlich die Geschwindigkeit beim initalen Start der App.

Redux

Anschließend stellen wir Ihnen das *Redux*-Pattern und das Framework *NgRx* vor. Sie erfahren, wie Sie mithilfe von Redux den Anwendungsstatus zentral und gut wartbar verwalten können.

NativeScript

Zum Schluss betrachten wir am praktischen Beispiel, wie wir *NativeScript* einsetzen können, um native mobile Anwendungen für verschiedene Zielplattformen (Android, iOS etc.) zu entwickeln.

Wissenswertes

Im letzten Kapitel des Buchs finden Sie weitere Informationen zu wissenswerten und begleitenden Themen. Hier haben wir weiterführende Inhalte zusammengetragen, auf die wir im Beispielprojekt nicht ausführlich eingehen.

Abtippen statt Copy & Paste

Wir alle kennen es: Beim Lesen steht vor uns ein großer Abschnitt Quelltext, und wir haben wenig Lust auf Tipparbeit. Schnell kommt der Gedanke auf, ein paar Codezeilen oder sogar ganze Dateien aus dem Repository zu kopieren. Vielleicht denken Sie sich: »Den Inhalt anzuschauen und die Beschreibung zu lesen reicht aus, um es zu verstehen.«

Abtippen heißt Lernen und Verstehen.

An dieser Stelle möchten wir aber einhaken. Kopieren und Einfügen ist nicht dasselbe wie *Lernen* und *Verstehen*. Wenn Sie die Codebeispiele selbst *eintippen*, werden Sie besser verstehen, wie Angular funktioniert, und werden die Software später erfolgreich in der Praxis einsetzen können. Jeder einzelne Quelltext, den Sie abtippen, trainiert Ihre Hände, Ihr Gehirn und Ihre Sinne. Wir möchten Sie deshalb ermutigen: Betrügen Sie sich nicht selbst. Der bereitgestellte Quelltext im Repository sollte lediglich der Überprüfung dienen. Wir wissen, wie schwer das ist, aber vertrauen Sie uns: Es zahlt sich aus, denn Übung macht den Meister!

Beratung und Workshops

Wir, die Autoren dieses Buchs, arbeiten seit Langem als Berater und Trainer für Angular. Wir haben die Erfahrung gemacht, dass man Angular in kleinen Gruppen am schnellsten lernen kann. In einem Workshop kann auf individuelle Fragen und Probleme direkt eingegangen werden – und es macht auch am meisten Spaß!

Schauen Sie auf https://angular.schule vorbei. Dort bieten wir Ihnen Angular-Workshops in den Räumen Ihres Unternehmens oder in offenen Gruppen an. Das Angular-Buch verwenden wir dabei in unseren Einstiegskursen zur Nacharbeit. Haben Sie das Buch vollständig gelesen, so können Sie direkt in die individuellen Kurse für Fortgeschrittene einsteigen. Wir freuen uns auf Ihren Besuch.

Die Angular.Schule: Workshops und Beratung

https://angular.schule

Danksagung

Dieses Buch hätte nicht seine Reife erreicht ohne die Hilfe und Unterstützung verschiedener Menschen. Wir danken **Gregor Woiwode** für die Mitwirkung als Autor in der ersten Auflage. Besonderer Dank geht an **Michael Kaaden** für seine unermüdlichen Anregungen und kritischen Nachfragen. Michael hat das gesamte Buch mehrfach auf Verständlichkeit und Fehler abgeklopft, alle Codebeispiele nachvollzogen und viel wertvollen Input geliefert. **Matthias Jauernig**, **Dilyana Pavlova**, **Silvio Böhme**, **Danilo Hoffmann** und **Jan-Niklas Wortmann** danken wir ebenso für die hilfreichen Anregungen und Korrekturvorschläge zur zweiten Auflage. **Julian Steiner** hat uns mit seiner Expertise zu NativeScript bei der Entwicklung der BookMonkey-Mobile-App unterstützt. Wertvolles Feedback zur ersten Auflage dieses Buchs haben uns außerdem **Nils Frohne**, **Johannes Hamfler**, **Stephan Hartmann**, **Johannes Hofmeister**, **Alexander Szczepanski** und **Daniel Vladut** zukommen lassen.

Dem Team vom dpunkt.verlag, insbesondere **René Schönfeldt** und **Sabrina Dietze**, danken wir für die persönliche Unterstützung und die guten Anregungen zum Buch. Außerdem danken wir dem **Angular-Team** und der Community dafür, dass sie eine großartige Plattform geschaffen haben, die uns den Entwickleralltag angenehmer macht.

Aktualisierungen in der zweiten Auflage

Die Webplattform bewegt sich schnell, und so muss auch ein Framework wie Angular stets an neue Gegebenheiten angepasst werden und mit den Anforderungen wachsen. In den zwei Jahren seit Veröffentlichung der ersten Auflage dieses Buchs haben sich viele Dinge geändert: Es wurden Best Practices etabliert, neue Features eingeführt, und einige wenige Features wurden wieder entfernt.

Die zweite Auflage, die Sie mit diesem Buch in Händen halten, wurde deshalb grundlegend aktualisiert und erweitert. Wir möchten Ihnen einen kurzen Überblick über die Neuerungen und Aktualisierungen der zweiten Auflage geben. Alle Texte und Codebeispiele haben wir auf die aktuelle Angular-Version aktualisiert.

Neue Kapitel

Folgende Kapitel und Abschnitte sind in dieser Auflage neu hinzugekommen:

- 10.3 Interceptoren: HTTP-Requests abfangen und transformieren (Seite 245)
- 19 Server-Side Rendering mit Angular Universal (Seite 537)
- 24 Wissenswertes (Seite 641)
  - 24.1 Container und Presentational Components (Seite 641)
  - 24.3 TrackBy-Funktion für die Direktive `ngFor` (Seite 646)
  - 24.4 Schematics: Codegenerierung mit der Angular CLI (Seite 648)
  - 24.6 Angular Material und weitere UI-Komponentensammlungen (Seite 654)
  - 24.11 Angular updaten (Seite 675)

Vollständig neu geschriebene Kapitel

Einige bereits in der ersten Auflage existierende Kapitel wurden für die zweite Auflage vollständig neu aufgerollt:

1 Schnellstart (Seite 3) Der Schnellstart basierte in der ersten Auflage auf einer lokalen Lösung mit SystemJS und Paketen aus einem CDN. Der neue Schnellstart setzt auf die Online-Plattform StackBlitz zum schnellen Prototyping von Webanwendungen.

10.2 Reaktive Programmierung mit RxJS (Seite 198) Das Prinzip der reaktiven Programmierung und das Framework RxJS haben in den letzten Jahren weiter an Bedeutung gewonnen. Das alte Kapitel zu RxJS lieferte nur einen kurzen Überblick, ohne auf Details einzugehen. Mit dieser Neufassung finden Sie jetzt eine ausführliche Einführung in die Prinzipien von reaktiver Programmierung und Observables, und es werden alle wichtigen Konzepte anhand von Beispielen erklärt. Im Gegensatz zur ersten Auflage verwenden alle Beispiele im Buch die aktuelle Syntax von RxJS mit den *Pipeable Operators*.

12 Formularverarbeitung & Validierung: Iteration IV (Seite 261) In der ersten Auflage haben wir sowohl *Template-Driven Forms* als auch *Reactive Forms* gleichbedeutend vorgestellt. Wir empfehlen mittlerweile nicht mehr den Einsatz von Template-Driven Forms. Daher stellen wir zwar beide Ansätze weiterhin vor, legen aber im Kapitel zur Formularverarbeitung einen stärkeren Fokus auf Reactive Forms. Das Praxisbeispiel wurde neu entworfen, um eine saubere Trennung der Zuständigkeiten der Komponenten zu ermöglichen. Die Erläuterungen im Grundlagenteil wurden neu formuliert, um besser für die Anforderungen aus der Praxis geeignet zu sein.

20 State Management mit Redux (Seite 553) In den letzten zwei Jahren hat sich unserer Ansicht nach das Framework *NgRx* gegen weitere Frameworks wie *angular-redux* klar durchgesetzt. Während die erste Auflage in diesem Kapitel noch auf angular-redux setzte, arbeitet das neue Kapitel durchgehend mit den *Reactive Extensions for Angular (NgRx)*. Das neue Kapitel erarbeitet in der Einführung schrittweise ein Modell für zentrales State Management, um die Architektur von Redux zu erläutern, ohne eine konkrete Bibliothek zu nutzen.

Stark überarbeitete und erweiterte Kapitel

4 Einführung in TypeScript (Seite 27) Das Grundlagenkapitel zu TypeScript wurde neu strukturiert und behandelt zusätzlich auch neuere Features von ECMAScript/TypeScript, z. B. Destrukturierung, Spread-Operator und Rest-Syntax.

10.1 HTTP-Kommunikation: ein Server-Backend anbinden (Seite 181) Das HTTP-Kapitel setzt durchgehend auf den `HttpClient`, der mit Angular 4.3 eingeführt wurde. Dabei wird der Blick auch auf die erweiterten Features des Clients geworfen. Themen, die spezifisch für RxJS sind, wurden aus diesem Kapitel herausgelöst und werden nun im RxJS-Kapitel behandelt.

14.4.1 Resolver: asynchrone Daten beim Routing vorladen (Seite 427) Resolver sind aus unserer Sicht nicht die beste Wahl, um reguläre Daten über HTTP nachzuladen. Der Iterationsschritt zu Resolvern wurde deshalb aus dem Beispielprojekt entfernt, und das Thema wird in dieser Auflage nur noch in der Theorie behandelt.

15.1 i18n: mehrere Sprachen und Kulturen anbieten (Seite 433) Die Möglichkeiten zur Konfiguration des Builds wurden mit Angular 6 stark vorangebracht. Viele zuvor notwendige Kommandozeilenparameter sind nun nicht mehr notwendig, die Konfigurationsdatei `angular.json` löst diese ab. Dadurch konnten wir das Kapitel zur Internationalisierung (i18n) kürzen und verständlicher gestalten. Im Gegensatz zur ersten Auflage zeigen wir nicht mehr, wie man eine Anwendung im JIT-Modus internationalisiert, der hauptsächlich für die Entwicklung vorgesehen ist, aber nicht für produktive Anwendungen.

17.1 Softwaretests (Seite 457) Das Kapitel zum Testen von Angular-Anwendungen wurde stark erweitert. Neben den reinen Werkzeugen wird der Fokus besonders auf Philosophien, Patterns und Herangehensweisen gelegt. Zusätzlich werden die mitgelieferten Tools zum Testen von HTTP und Routing betrachtet.

22 NativeScript: mobile Anwendungen entwickeln (Seite 601) Zur Entwicklung einer nativen mobilen Anwendung nutzen wir in diesem Kapitel die neue Version 3 von NativeScript, die insbesondere Verbesserungen in Sachen Codegenerierung und Wiederverwendbarkeit von Code mitbringt.

24.9 Change Detection (Seite 661) Das Kapitel zur Change Detection wurde für besseres Verständnis neu strukturiert. Insbesondere wird auf Debugging und Strategien zur Optimierung eingegangen.

Sonstiges

Für die einzelnen Iterationsschritte aus dem Beispielprojekt bieten wir eine Differenzansicht an. So können die Änderungen am Code zwischen den einzelnen Kapiteln besser nachvollzogen werden. Wir gehen darauf auf Seite 49 genauer ein.

Zu guter Letzt haben wir an ausgewählten Stellen in diesem Buch Zitate von bekannten Persönlichkeiten aus der Angular-Community aufgeführt. Die meisten dieser Zitate haben wir direkt für dieses Buch erbeten. Wir freuen uns sehr, dass so viele interessante und humorvolle Worte diesem Buch eine einmalige Note geben.

Inhaltsverzeichnis

Teil II

TypeScript

4 Einführung in TypeScript

»In any modern frontend project, TypeScript is an absolute no-brainer to me. No types, no way!«

Marius Schulz
(Front End Engineer und Trainer für JavaScript)

Für die Entwicklung mit Angular werden wir die Programmiersprache *TypeScript* verwenden. Doch keine Angst – Sie müssen keine neue Sprache erlernen, um mit Angular arbeiten zu können, denn TypeScript ist eine Obermenge von JavaScript.

Wenn Sie bereits erste Erfahrungen mit TypeScript gemacht haben, können Sie dieses Kapitel überfliegen oder sogar überspringen. Viele Eigenheiten werden wir auch auf dem Weg durch unsere Beispielanwendung kennenlernen. Wenn Sie unsicher sind oder TypeScript und modernes JavaScript für Sie noch Neuland sind, dann ist dieses Kapitel das Richtige für Sie. Wir wollen in diesem Kapitel die wichtigsten Features von TypeScript erläutern, damit Sie für den Einstieg in Angular bestens gewappnet sind. Wir werden die wichtigsten Sprachelemente von TypeScript kennenlernen, sodass es uns im weiteren Verlauf des Buchs leichtfällt, die gezeigten Codebeispiele zu verstehen.

Sie können dieses Kapitel später als Referenz verwenden, wenn Sie mit TypeScript einmal nicht weiterwissen. *Auf geht's!*

4.1 Was ist TypeScript und wie setzen wir es ein?

TypeScript ist eine Obermenge von JavaScript. Die Sprache greift die aktuellen ECMAScript-Standards auf und integriert zusätzliche Features, unter anderem ein statisches Typsystem. Das bedeutet allerdings nicht, dass Sie eine komplett neue Programmiersprache lernen müssen. Ihr bestehendes Wissen zu JavaScript bleibt weiterhin anwendbar, denn TypeScript erweitert lediglich den existierenden Sprachstandard. Jedes Programm, das in JavaScript geschrieben wurde, funktioniert auch in TypeScript.

TypeScript integriert Features aus kommenden JavaScript-Standards.

TypeScript unterstützt neben den existierenden JavaScript-Features auch Sprachbestandteile aus zukünftigen Standards. Das hat den Vorteil, dass wir das Set an Sprachfeatures genau kennen und alle verwendeten Konstrukte in allen gängigen Browsern unterstützt werden. Wir müssen also nicht lange darauf warten, dass ein Sprachfeature irgendwann einmal direkt vom Browser unterstützt wird, und können stattdessen sofort loslegen. Zusätzlich bringt TypeScript ein statisches Typsystem mit, mit dem wir schon zur Entwicklungszeit eine hervorragende Unterstützung durch den Editor und das Build-Tooling genießen können.

TypeScript-Compiler

TypeScript ist nicht im Browser lauffähig, denn zusammen mit dem Typsystem und neuen Features handelt es sich nicht mehr um reines JavaScript. Deshalb wird der TypeScript-Code vor der Auslieferung wieder in JavaScript umgewandelt. Für diesen Prozess ist der TypeScript-Compiler verantwortlich. Man spricht dabei auch von *Transpilierung*, weil der Code lediglich in eine andere Sprache übertragen wird. Alle verwendeten Sprachkonstrukte werden so umgewandelt, dass sie dieselbe Semantik besitzen, aber nur die Mittel nutzen, die tatsächlich von JavaScript in der jeweiligen Version unterstützt werden. Die statische Typisierung geht bei diesem Schritt verloren. Das bedeutet, dass das Programm zur Laufzeit keine Typen mehr besitzt, denn es ist ein reines JavaScript-Programm. Durch die Typunterstützung bei der Entwicklung und beim Build können allerdings schon die meisten Fehler erschlagen werden.

Abbildung 4–1 zeigt, wie TypeScript die bestehenden JavaScript-Versionen erweitert. Der Standard ECMAScript 2016 hat keine nennenswerten Features gebracht, sodass dieser nicht weiter erwähnt werden muss. TypeScript vereint viele Features aus aktuellen und kommenden ECMAScript-Versionen, sodass wir sie problemlos auch für ältere Browser einsetzen können.

Verwirrung um die ECMAScript-Versionen

Der JavaScript-Sprachkern wurde über viele Jahre hinweg durch die European Computer Manufacturers Association (ECMA) weiterentwickelt. Dabei wurden die Versionen zunächst fortlaufend durchnummeriert: ES1, ES2, ES3, ES4, ES5.

Noch während die nächste Version spezifiziert wurde, war bereits der Name *ES6* in aller Munde. Kurz vor Veröffentlichung des neuen Sprachstandards entschied sich jedoch das technische Komitee der ECMA dazu, eine neue Namenskonvention einzuführen. Da fortan jährlich eine neue Version erscheinen soll, wurde die Bezeichnung vom schon vielerseits etablierten *ES6* zu *ECMAScript 2015* geändert.

Aufgrund der parallelen Entwicklung vieler Polyfills und Frameworks findet man in einschlägiger Literatur und auch in vielen Entwicklungsprojekten noch die Bezeichnung *ES6*.

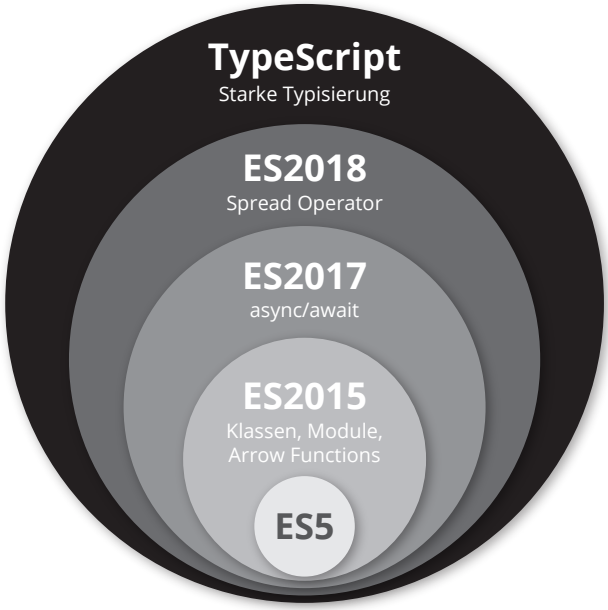


Abb. 4–1
TypeScript und ECMAScript

```
8   if (environment.production) {
9     enableProdMode();
10  }
11
12  platformBrowserDynamic().bootstrapModule(AppModule);
13
```

```
(method) PlatformRef.bootstrapModule<AppModule>(moduleType: Type<AppModule>, compilerOptions?: CompilerOptions | CompilerOptions[]): Promise<NgModuleRef<AppModule>>

Creates an instance of an @NgModule for a given platform using the given runtime compiler.
```

Abb. 4–2
Unterstützung durch den Editor: Type Information On Hover

Typisierung

TypeScript ist als Open-Source-Projekt bei der Firma Microsoft entstanden.[1] Durch die Typisierung können Fehler bereits zur Entwicklungszeit erkannt werden. Außerdem können Tools den Code genauer analysieren. Dies ermöglicht Komfortfunktionen wie automatische Vervollständigung, Navigation zwischen Methoden und Klassen, eine solide Refactoring-Unterstützung und automatische Dokumentation in der Entwicklungsumgebung. TypeScript kompiliert in reines JavaScript (unter anderem nach ES5) und ist dadurch in allen Browsern und auf allen Plattformen ausführbar.

Neue JavaScript-Features auf einen Blick

Bei Interesse können Sie mithilfe einer Kompatibilitätstabelle[2] einen guten Überblick erhalten, welche Features der verschiedenen Standards bereits implementiert wurden.

[1] https://ng-buch.de/a/26 – TypeScript

[2] https://ng-buch.de/a/27 – ECMAScript compatibility table

Teil III

BookMonkey 3:
Schritt für Schritt zur App

6 Komponenten & Template-Syntax: Iteration I

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

Igor Minar
(Angular Lead Developer)

Nun, da unsere Projektumgebung vorbereitet ist, können wir beginnen, die ersten Schritte bei der Implementierung des BookMonkeys zu machen. Wir wollen in dieser Iteration die wichtigsten Grundlagen von Angular betrachten. Wir lernen zunächst das Prinzip der komponentenbasierten Entwicklung kennen und tauchen in die Template-Syntax von Angular ein. Zwei elementare Konzepte – die Property und Event Bindings – schauen wir uns dabei sehr ausführlich an.

Die Grundlagen von Angular sind umfangreich, deshalb müssen wir viel erläutern, bevor es mit der Implementierung losgeht. Aller Anfang ist schwer, aber haben Sie keine Angst: Sobald Sie die Konzepte verinnerlicht haben, werden sie Ihnen den Entwickleralltag angenehmer machen!

6.1 Komponenten: die Grundbausteine der Anwendung

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

6.1.1 Komponenten

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

Hauptkomponente

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

Eine Komponente besitzt immer ein Template.

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

Komponente: Klasse mit Decorator @Component()

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

Was ist ein Decorator?

Decorators dienen der Angabe von Metainformationen zu einer Komponente. Der Einsatz von Metadaten fördert die Übersichtlichkeit im Code, da Konfiguration und Ablaufsteuerung sauber voneinander getrennt werden. Wer mit der Verwendung von Decorators noch nicht vertraut ist, sollte sich den Abschnitt »Decorators« auf Seite 46 durchlesen.

Listing 6–1 *Eine simple Komponente*

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

Metadaten

Selektor

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

Was ist ein CSS-Selektor?

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

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

Tab. 6–1 *Geläufige CSS-Selektoren*

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

Host-Element

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

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

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

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

deren einsetzen usw. Diese Praxis schauen wir uns im nächsten Kapitel ab Seite 97 genauer an.

Komponenten sollten nur auf Elementnamen selektieren.

Es ist eine gute Praxis, stets nur Element*namen* zu verwenden, um Komponenten einzubinden. Das Prinzip der Komponente – Template und angeheftete Logik – kann durch ein eigenständiges Element am sinnvollsten abgebildet werden. Wenn wir auf die Attribute eines Elements selektieren wollen, so sind *Attributdirektiven* ein sinnvoller Baustein. Wie das funktioniert und wie wir eigene Direktiven implementieren können, schauen wir uns ab Seite 365 an.

Das Template einer Komponente

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

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

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

Eine Komponente besitzt genau ein Template.

In beiden Fällen verwenden wir die Metadaten des `@Component()`-Decorators, um die Infos anzugeben. Im Listing 6–3 sind beide Varianten zur Veranschaulichung aufgeführt. Es kann allerdings immer nur einer der beiden Wege verwendet werden, denn eine Komponente besitzt nur ein einziges Template. Die Angular CLI legt stets eine separate Datei für das Template an, sofern wir es nicht anders einstellen. Das ist auch die Variante, die wir Ihnen empfehlen möchten, um die Struktur übersichtlich zu halten.

[1] Später im Kapitel zu NativeScript (ab Seite 601) werden wir einen Einsatzzweck ohne HTML kennenlernen.

Listing 6–3
Template einer Komponente definieren

```
@Component({

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

  // ODER: als HTML-String direkt im TypeScript
  template: `<h1>
          Hallo Angular!
  </h1>`,

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

Bindings für die Kommunikation zwischen Komponente und Template

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

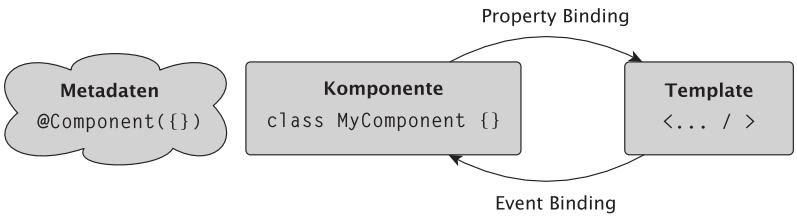


Abb. 6–1
Komponente, Template und Bindings im Zusammenspiel

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

Der Style einer Komponente

Um das Aussehen einer Komponente zu beeinflussen, werden Stylesheets eingesetzt, wie wir sie allgemein aus der Webentwicklung kennen. Neben den reinen Cascading Style Sheets (CSS) können wir auch verschiedene Präprozessoren einsetzen: Sass, Less und Stylus werden direkt unterstützt. Welches Style-Format wir standardmäßig verwenden wollen, können wir auswählen, wenn wir die Anwendung mit `ng new` erstellen.

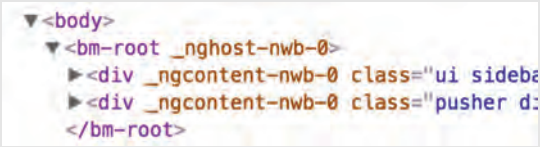
Normalerweise verwendet man eine große, globale Style-Datei, die alle gestalterischen Aspekte der Anwendung definiert. Das ist nicht immer schön, denn hier kann man leicht den Überblick verlieren, welche Selektoren wo genau aktiv sind oder womöglich gar nicht mehr benötigt werden. Außerdem widerspricht eine globale Style-Definition dem modularen Prinzip der Komponenten.

Stylesheets von Komponenten sind isoliert.

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

Ein Blick ins Innere: View Encapsulation

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



Angular generiert automatisch Attribute für die View Encapsulation.

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

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

Im Listing 6–4 werden beide Wege gezeigt. Wichtig ist, dass die Dateien und Styles jeweils als Arrays angelegt werden. Grundsätzlich empfehlen wir Ihnen auch hier, für die Styles eine eigene Datei anzulegen und in

der Komponente zu referenzieren. Die Angular CLI unterstützt beide Varianten.

Der herkömmliche Weg zum Einbinden von Styles ist natürlich trotzdem weiter möglich: Wir können globale CSS-Dateien definieren, die in der gesamten Anwendung gelten und nicht nur auf Ebene der Komponenten. Diesen Weg haben wir gewählt, um das Style-Framework Semantic UI einzubinden, siehe Seite 65.

Listing 6–4
Style-Definitionen in Komponenten

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

6.1.2 Komponenten in der Anwendung verwenden

Rule of One

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

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

Komponenten im AppModule *registrieren*

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

Dazu dient die Eigenschaft `declarations` im Decorator `@NgModule()`. Hier werden alle Komponenten[2] notiert, die zur Anwendung gehören. Damit wir die Typen dort verwenden können, müssen wir alle Komponenten importieren.

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

Demo und Quelltext:
https://ng-buch.de/bm3-it3-http

10.2 Reaktive Programmierung mit RxJS

»RxJS is one of the best ways to utilize reactive programming practices within your codebase. By starting to think reactively and treating everything as sets of values, you'll start to find new possibilities of how to interact with your data within your application.«

Tracy Lee
(Google Developer Expert und Mitglied im RxJS Core Team)

Reaktive Programmierung ist ein Programmierparadigma, das in den letzten Jahren verstärkt Einzug in die Welt der Frontend-Entwicklung gehalten hat. Die mächtige Bibliothek *Reactive Extensions für JavaScript (RxJS)* greift diese Ideen auf und implementiert sie. Der wichtigste Datentyp von RxJS ist das Observable – ein Objekt, das einen Datenstrom liefert. Tatsächlich dreht sich die Idee der reaktiven Programmierung im Wesentlichen darum, Datenströme zu verarbeiten und auf Veränderungen zu reagieren. Wir haben in diesem Buch bereits mit Observables gearbeitet, ohne näher darauf einzugehen. Da Angular an vielen Stellen auf RxJS setzt, wollen wir einen genaueren Blick auf das Framework und die ihm zugrunde liegenden Prinzipien werfen.

10.2.1 Alles ist ein Datenstrom

Bevor wir damit anfangen, uns mit den technischen Details von RxJS auseinanderzusetzen, wollen wir uns mit der Grundidee der reaktiven Programmierung befassen: *Datenströme*. Wenn wir diesen Begriff ganz untechnisch betrachten, so können wir das Modell leicht auf die alltägliche Welt übertragen. Unsere gesamte Interaktion und Kommunikation mit der Umwelt basiert auf Informationsströmen.

Der Wecker klingelt.

Das beginnt bereits morgens vor dem Aufstehen: Der Wecker klingelt (Ereignis), Sie reagieren darauf und drücken die Schlummertaste. Nach 10 Minuten klingelt der Wecker wieder, und Sie stehen auf.[3]

[3] Wir gehen natürlich davon aus, dass Sie die Schlummerfunktion mehr als einmal benutzen, aber für das Beispiel soll es so genügen.

Sie haben einen Strom von wiederkehrenden Ereignissen abonniert und verändern den Datenstrom mithilfe von Aktionen. Schon dieser Ablauf ist von vielen Variablen und Entscheidungen geprägt: Wie viel Zeit habe ich noch? Was muss ich noch erledigen? Fühle ich mich wach oder möchte ich weiterschlafen?

Warten auf den Bus

Sie gehen aus dem Haus und warten auf den Bus. Damit Sie in den richtigen Bus steigen, ignorieren Sie zunächst alle anderen Verkehrsmittel, bis der Bus auf der Straße erscheint – Sie haben also einen Strom von Verkehrsmitteln beobachtet, das passende herausgesucht und damit interagiert. Dafür haben Sie eine konkrete Regel angewendet: Ich benötige den Bus der Linie 70.

Telefonieren und Chatten

Im Bus klingelt das Telefon, Sie heben ab und sprechen mit dem Anrufer. Beide Teilnehmer erzeugen einen Informationsstrom und reagieren auf die ankommenden Informationen. Aus einigen Teilen des Gesprächs leiten Sie konkrete Aktionen ab (z. B. antworten oder etwas erledigen), andere Teile sind unwichtig. Während Sie telefonieren, vibriert das Handy, denn Sie haben eine Chatnachricht erhalten. Und noch eine. Und noch eine. Die Nachrichten treffen nacheinander ein – Sie ignorieren die Ereignisse allerdings, denn das Chatprogramm puffert die Nachrichten, sodass Sie den Text auch später lesen können. Später sehen Sie, dass die Nachrichten von verschiedenen Personen stammen: Einzelne Menschen haben Nachrichten erzeugt, die bei Ihnen in einem großen Datenstrom eingetroffen sind.

Frühstück kaufen

Nach dem Aussteigen holen Sie sich beim Bäcker etwas zu essen: Sie gehen in den Laden, beobachten den Datenstrom von Angeboten in der Theke, wählen ein Angebot aus und starten den Kaufvorgang. Schließlich verlassen Sie das Geschäft mit einem Brötchen. Was ist passiert? Ein Strom von eingehenden Kunden, die Geld besitzen, wurde umgewandelt in einen Strom von ausgehenden Kunden, die nun Brötchen haben. Der Angestellte beim Bäcker hat den Kundenstrom abonniert und die einzelnen Elemente mit Backwaren versorgt.

Wir könnten dieses Beispiel beliebig weiterführen, aber der Kern der Idee ist bereits erkennbar: Das komplexe System in unserer Welt basiert darauf, dass Ereignisse auftreten, auf die wir reagieren können. Durch unsere Erfahrung wissen wir, wie mit bestimmten Ereignissen umzugehen ist, z. B. wissen wir, wie man ein Telefon bedient, ein Gespräch führt oder ein Brötchen kauft. Manche Ereignisse treten nur für uns und als Folge anderer Aktionen auf: Das Brötchen wird erst eingepackt, wenn wir es kaufen. Lösen wir die Aktion nicht aus, so findet kein Ereignis statt. Andere Ereignisse hingegen passieren auch ohne dass wir darauf einen Einfluss haben, z. B. der Straßenverkehr oder das Wetter. Unsere Aufgabe ist es, diese Ereignisse zu beobachten und pas-

send darauf zu reagieren. Sind wir nicht an den Ereignissen interessiert, passieren sie trotzdem.

Ereignisse in der Software

Die Aufgabe von Software ist es, Menschen in ihren Aufgaben und Abläufen zu unterstützen. Daher finden wir viele Ansätze aus der echten Welt eben auch in der Softwareentwicklung wieder. Unsere Anwendungen sind von einer Vielzahl von Ereignissen und Einflüssen geprägt: Der Nutzer interagiert mit der Anwendung, klickt auf Buttons und füllt Formulare aus. API-Requests kommen vom Server zurück und Timer laufen ab. Wir möchten auf all diese Ereignisse passend reagieren und weitere Aktionen anstoßen. Wenn Sie einmal an eine interaktive Anwendung wie Tabellenkalkulation denken, wird dieses Prinzip deutlich: Sie füllen ein Feld aus, das Teil einer komplexen Formel ist, und alle zugehörigen Felder werden automatisch aktualisiert.

Alles ist ein Datenstrom.

Datenströme verarbeiten, zusammenführen, transformieren und filtern – das ist die Grundidee der reaktiven Programmierung. Das Modell geht davon aus, dass sich alles als ein Datenstrom auffassen lässt: nicht nur Ereignisse, sondern auch Variablen, statische Werte, Nutzereingaben und vieles mehr. Zusammen mit den Ideen aus der funktionalen Programmierung ergibt sich aus dieser Denkweise eine Vielzahl von Möglichkeiten, um Programmabläufe und und Veränderungen an Daten *deklarativ* zu modellieren.

10.2.2 Observables sind Funktionen

Um die Idee der allgegenwärtigen Datenströme in unserer Software aufzugreifen, benötigen wir zuerst ein Konstrukt, mit dem sich ein Datenstrom abbilden lässt. Wir wollen ein Objekt entwerfen, das über die Zeit nacheinander mehrere Werte ausgibt. Jeder, der an den Werten interessiert ist, kann den Datenstrom abonnieren. Dabei soll es drei Arten von Ereignissen geben:

- Ein neues Element trifft ein (`next`).
- Ein Fehler tritt auf (`error`).
- Der Datenstrom ist planmäßig zu Ende (`complete`).

Wir erstellen dazu eine einfache JavaScript-Funktion mit dem Namen `observable()`, die wir im weiteren Verlauf als *Producer*-Funktion bezeichnen wollen. Als Argument erhält diese Funktion ein Objekt, das drei Eigenschaften mit *Callback*-Funktionen besitzt: `next`, `error` und `complete`. Dieses Objekt nennen wir *Observer*. Im Körper der Producer-Funktion führen wir nun beliebige Aktionen aus, so wie es eben für eine Funktion üblich ist. Immer wenn etwas passiert, rufen wir eins der drei Callbacks aus dem Observer auf: Wenn ein neuer Wert ausgegeben werden soll, wird `next()` gerufen, sind alle Aktionen abgeschlossen,

rufen wir `complete()` auf, und tritt ein Fehler in der Verarbeitung auf, so nutzen wir `error()`. Diese Aufrufe können synchron oder zeitversetzt erfolgen. Welche Aktionen wir hier ausführen, ist ganz unserer konkreten Implementierung überlassen.

Listing 10–17
Producer-Funktion

```
function observable(observer) {
  setTimeout(() => {
    observer.next(1);
  }, 1000);

  observer.next(2);

  setTimeout(() => {
    observer.next(3)
    observer.complete();
  }, 2000);
}
```

Damit in unserem Programm auch tatsächlich etwas passiert, rufen wir die Producer-Funktion `observable()` auf und übergeben als Argument einen konkreten Observer, also ein Objekt mit den drei Callbacks `next`, `error` und `complete`.

Listing 10–18
Funktion mit Observer aufrufen

```
const myObserver = {
  next:  value => console.log('NEXT:', value),
  error: err => console.log('ERROR:', err),
  complete: () => console.log('COMPLETE')
};

observable(myObserver);
```

Das Programm erzeugt die folgende zeitversetzte Ausgabe, die sich auch auf einem Zeitstrahl darstellen lässt.

Listing 10–19
Ausgabe des Programms

```
NEXT: 2
NEXT: 1
NEXT: 3
COMPLETE
```

Abb. 10–1
Grafische Darstellung der Ausgabe

Reduzieren wir diese Idee auf das Wesentliche, so lässt sie sich wie folgt zusammenfassen: Wir haben eine Funktion entwickelt, die Befehle ausführt und ein Objekt entgegennimmt, das drei Callback-Funktionen enthält. Wenn im Programmablauf etwas passiert (synchron oder asynchron), wird eines dieser drei Callbacks aufgerufen. Die Producer-Funktion emittiert also nacheinander verschiedene Werte an den Observer.

Immer wenn die Funktion `observable()` aufgerufen wird, startet das Programm. Daraus folgt, dass nichts passiert, wenn niemand die Funktion aufruft. Starten wir die Funktion hingegen mehrfach, so werden die Routinen auch mehrfach ausgeführt. Was zunächst ganz offensichtlich klingt, ist eine wichtige Eigenschaft, auf die wir später noch zurückkommen werden.

An dieser Stelle möchten wir Sie aber zunächst beglückwünschen! Wir haben gemeinsam unser erstes *Observable* entwickelt und haben dabei gesehen: Ein Observable ist nichts anderes als eine Funktion.

10.2.3 Das Observable aus RxJS

ReactiveX, auch *Reactive Extensions* oder kurz *Rx* genannt, ist ein reaktives Programmiermodell, das ursprünglich von Microsoft für das .NET-Framework entwickelt wurde. Die Implementierung ist sehr gut durchdacht und verständlich dokumentiert. Die Idee erfreut sich großer Beliebtheit, und so sind sehr viele Portierungen für die verschiedensten Programmiersprachen entstanden. Der wichtigste Datentyp von Rx, das *Observable*, ist sogar mittlerweile ein Vorschlag für ECMAScript[4] geworden. RxJS ist der Name der JavaScript-Implementierung von ReactiveX.

Angular setzt intern stark auf die Möglichkeiten von RxJS, einige haben wir sogar schon kennengelernt: Der `EventEmitter` ist ein Observable, der `HttpClient` gibt Observables zurück und auch Formulare und der Router propagieren Änderungen mit Observables.

Die Observable-Implementierung von RxJS folgt genau der Idee, die wir im letzten Abschnitt an unserem Funktionsbeispiel entwickelt haben: Wir rufen eine Funktion auf, übergeben einen Observer, und die Funktion ruft die Callbacks aus dem Observer auf, sobald etwas passiert. RxJS bringt für sein Observable allerdings zusätzlich einen wohldefinierten Rahmen mit und befolgt einige Regeln. Dazu gehören unter anderem folgende Punkte:

[4] https://ng-buch.de/a/45 – GitHub: TC39 Observables for ECMAScript

- Der Datenstrom ist zu Ende, sobald `error()` oder `complete()` gerufen wurden. Es ist also nicht möglich, danach noch einmal reguläre Werte mit `next()` auszugeben.
- Das Observable besitzt die Methode `subscribe()`, mit der wir den Datenstrom abonnieren können. Abonnierte Daten können außerdem wieder abbestellt werden.
- Ein Observable besitzt die Methode `pipe()`. Damit können wir sogenannte Operatoren an das Observable anhängen, um den Datenstrom zu verändern.
- Der fest definierte Datentyp `Observable` sorgt dafür, dass Observables aus verschiedenen Quellen miteinander kompatibel sind.

Wir werden den Aufbau und die Funktionsweise eines solchen Observables in den folgenden Abschnitten genauer betrachten. Behalten Sie dabei das Funktionsbeispiel im Hinterkopf, denn Sie werden einige Dinge wiedererkennen.

10.2.4 Observables abonnieren

Um die Daten aus einem Observable zu erhalten, müssen wir den Datenstrom abonnieren. Da wir nun ein »echtes« Observable nutzen, funktioniert dieser Aufruf ein wenig anders als in unserem einfachen Funktionsbeispiel – hat aber starke Ähnlichkeiten. Jedes Observable besitzt eine Methode mit dem Namen `subscribe()`. Rufen wir sie auf, wird die Routine im Observable gestartet und das Objekt kann Werte ausgeben.

Als Argument übergeben wir ein Objekt mit drei Callback-Funktionen `next`, `error` und `complete`. Erkennen Sie die Parallelen? Diesen Observer haben wir im vorherigen Beispiel bereits verwendet. Das Observable ruft die drei Callbacks aus dem Observer auf und liefert auf diesem Weg Daten an den Aufrufer.

Listing 10–20
Observable abonnieren mit Observer

```
const myObserver = {
  next:  value => console.log('NEXT:', value),
  error: err => console.error('ERROR:', err),
  complete: () => console.log('COMPLETE')
};

myObservable$.subscribe(myObserver);
```

Neben dieser etwas aufwendigen Schreibweise gibt es noch einen anderen Weg. Anstatt ein Objekt mit drei Funktionen zu notieren, können wir die drei Callbacks auch einzeln nacheinander als Argumente von `subscribe()` angeben. Das hat den Vorteil, dass wir nicht immer

alle drei Funktionen übergeben müssen: Sind wir nur an den regulären Werten aus dem Observable interessiert, so reicht es aus, wenn wir das erste Callback notieren. Diese Schreibweise ist auch der normale und empfohlene Weg, den wir bereits im Kapitel zu HTTP verwendet haben.

Listing 10–21 *Observable abonnieren mit Callbacks*

```
// mit drei Callbacks
myObservable$.subscribe(
  value => console.log('NEXT:', value),
  err => console.error('ERROR:', err),
  () => console.log('COMPLETE')
);

// mit einem Callback
myObservable$
  .subscribe(value => console.log('NEXT:', value));
```

Subscriptions beenden

Sobald wir Daten von einem Observable abonnieren, gehen wir einen Vertrag ein: Das Observable liefert Daten, wir nehmen die Daten entgegen. Wie bei einem ordentlichen Vertrag üblich, lässt sich auch dieser von beiden Seiten kündigen. Beendet das Observable den Datenstrom (also durch `error` oder `complete`), so wird auch die Subscription automatisch beendet.

Aber auch der Programmablauf außerhalb des Observables kann das Abonnement kündigen. Die Methode `subscribe()` gibt dazu ein Objekt vom Typ `Subscription` zurück. Dieses Objekt wiederum besitzt eine Methode `unsubscribe()`, mit der wir das Abonnement beenden können.

Listing 10–22 *Subscription beenden mit* unsubscribe()

```
const subscription = myObservable$
  .subscribe(myObserver);

setTimeout(() => {
  subscription.unsubscribe();
}, 3000);
```

Memory Leaks vermeiden

Das Observable liefert danach keine Daten mehr, und der Speicher wird wieder freigegeben. Das ist besonders wichtig, um Memory Leaks zu vermeiden, bei denen Speicher länger reserviert wird als nötig. Mit diesem Thema werden wir uns demnächst noch beschäftigen.

11 Powertipp: Augury

Bei der Entwicklung mit Angular ist es oft hilfreich, Informationen über den aktuellen Status unserer Anwendung zu erhalten. Augury[1] ist ein nützliches Werkzeug, mit dem wir die Anwendung zur Laufzeit untersuchen und debuggen können.

Die wichtigsten Funktionen von Augury sind:

- Beziehungen zwischen Komponenten darstellen (Komponentenhierarchie)
- detaillierte Informationen zu Komponenten anzeigen
- Abhängigkeiten darstellen (Injector Graph)
- Informationen zur Change Detection anzeigen
- den gesamten Routenbaum darstellen
- Zustände und Werte anzeigen und modifizieren
- Events auslösen

Installation

Das Tool ist als Erweiterung für Google Chrome verfügbar und kann über den Extension Manager des Browsers installiert werden.[2]

Abb. 11–1
Erweiterung Augury in Google Chrome

Nach der Installation taucht das Tool als zusätzlicher Reiter in den Chrome Developer Tools auf.

[1] https://ng-buch.de/a/15 – Angular Augury
[2] https://ng-buch.de/a/58 – Chrome Web Store: Augury

Der Component Tree

Beim ersten Aufruf von Augury gelangt man direkt zur Ansicht *Component Tree* (Abbildung 11–2). Diese Ansicht stellt eine Übersicht der Komponenten und deren Verschachtelung dar. Wenn wir mit dem Mauszeiger über einen Eintrag im *Component Tree* fahren, wird die entsprechende Komponente im Browser hervorgehoben. Wählen wir eine Komponente aus, so erscheinen im rechten Bereich weitere Details dieser Komponente. Es werden gesetzte *Propertys* und ggf. abhängige Provider dargestellt.

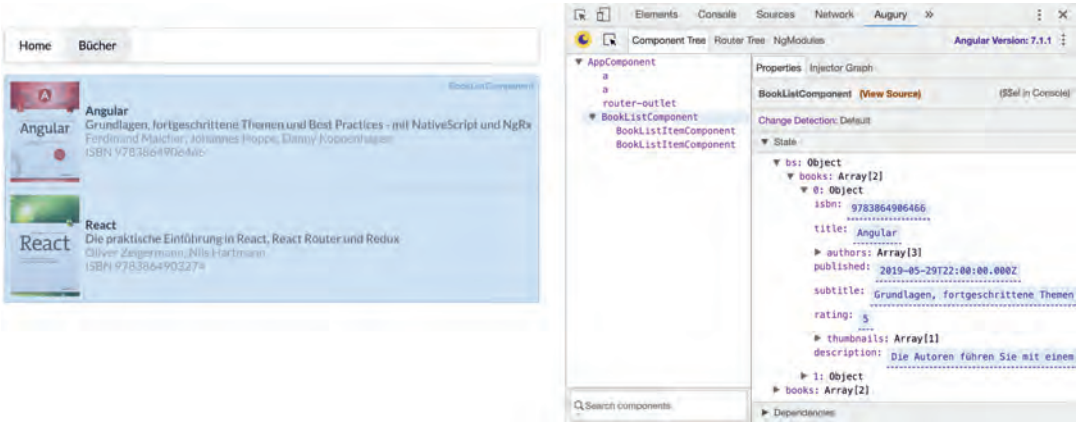


Abb. 11–2 *Eigenschaften einer Komponente mit Augury untersuchen*

Auf der rechten Seite ist außerdem der Reiter *Injector Graph* zu sehen. Hier werden die Abhängigkeiten einer Komponente grafisch dargestellt. Die Propertys lassen sich direkt über Augury editieren. Ändern wir einen Wert, so sehen wir sofort die Auswirkungen der Änderung. Ebenso können wir Events über eine entsprechende Eingabe und Betätigung des Buttons *Emit* auslösen.

Führen wir in der Anwendung eine Aktion aus (z. B. durch Klicken eines Buttons), hebt Augury alle betreffenden Komponenten farblich hervor, in denen der Change-Detection-Prozess angestoßen wird.

Im unteren Teil des Tools befindet sich ein Suchfeld, mit dem wir den gesamten Komponentenbaum nach einer bestimmten Komponente oder Attributen durchsuchen können.

Der Router Tree

Das zweite große Feature von Augury ist der Reiter *Router Tree*. Hier werden alle verfügbaren Routen und deren zugehörige Komponenten grafisch dargestellt. Es ist direkt ersichtlich, an welcher Stelle innerhalb des *Router Trees* die Anwendung sich gerade befindet und welche Routen von dort aus erreichbar sind.

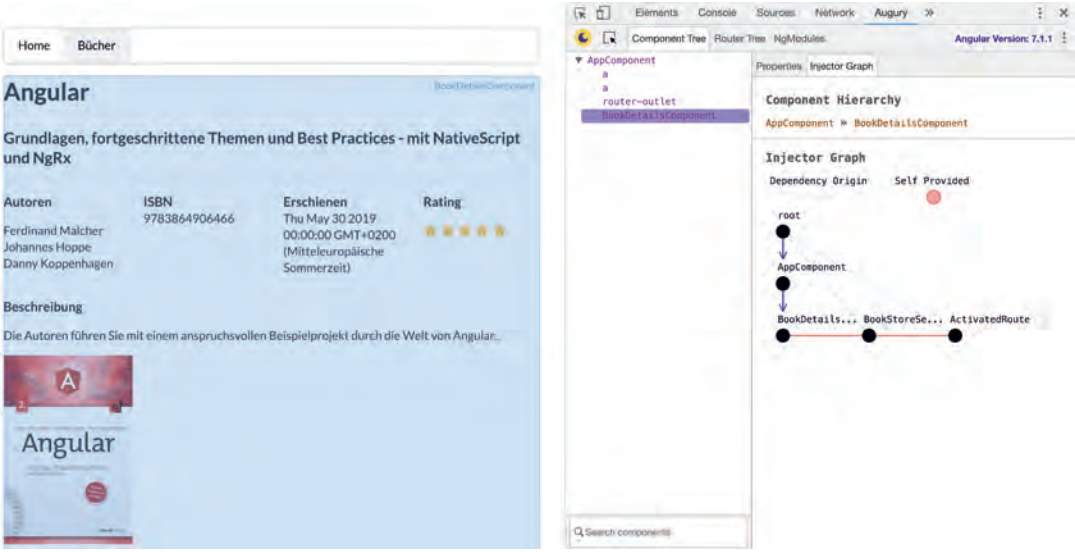


Abb. 11–3
Der Injector Graph von Augury

Wählen wir einen Knoten im Baum aus, werden der Pfad, die zugehörige Komponente und ggf. übergebene Daten angezeigt. Auch der Router Tree lässt sich nach Inhalten durchsuchen.

Zusammenfassung

Wir können mithilfe von Augury einen Überblick über unsere Komponenten, Services und Routen erlangen. Wir sehen, welche Abhängigkeiten diese zueinander haben, und können das Zusammenspiel bei Statusänderungen verfolgen. Augury kann beim Debugging einer Angular-Anwendung helfen. Wir können direkt im Tool Werte manipulieren, Events auslösen und sehen, wie unsere Anwendung darauf reagiert.

Teil V

Weiterführende Themen

20 State Management mit Redux

»NgRx provides robust state management for small and large projects. It enforces proper separation of concerns. Using it from the start reduces the risk of spaghetti when the project evolves.«

Minko Gechev
(Mitglied des Angular-Teams)

Wir haben in diesem Buch gelernt, wie wir eine Angular-Anwendung entwickeln, und haben dabei alle wichtigen Konzepte betrachtet. Unsere Anwendung haben wir komponentenzentriert und serviceorientiert aufgebaut: Die Komponenten unserer Anwendung kommunizieren auf klar definierten Wegen über Property Bindings und Event Bindings. Um Daten zu erhalten und zu senden, nutzen die Komponenten verschiedene Services, in denen die HTTP-Kommunikation gekapselt ist oder über die wir Daten austauschen können.

Diese Herangehensweise funktioniert im Prinzip sehr gut, und wir haben so eine vollständige Anwendung entwickeln können. Unsere Beispielanwendung ist allerdings auch recht klein und übersichtlich – in der Praxis werden die Anwendungen hingegen wesentlich größer: Viele Komponenten greifen dann gleichzeitig auf geteilte Daten zu und wollen dieselben Services nutzen. Auch die Performance spielt eine immer größere Rolle, je komplexer die Anwendung wird. Wir erreichen mit der bisher vorgestellten Herangehensweise schnell einen Punkt, an dem wir den Überblick über die Kommunikationswege verlieren. Es kommt immer häufiger zu unerklärlichen Konstellationen, da man nicht mehr nachvollziehen kann, welche Komponente andere Komponenten bzw. Services aufruft und in welcher Reihenfolge dies geschieht. Gleichzeitig führen die vielen Kommunikationswege zu entsprechend vielen Änderungen an den Daten, die von der Change Detection[1] erkannt und verarbeitet werden müssen. Kurzum: Die Anwendung wird zunehmend schwerfälliger.

[1] Die Ideen und Hintergründe der Change Detection besprechen wir ausführlich im Kapitel »Wissenswertes« ab Seite 661.

Mit wachsender Größe der Anwendung ergeben sich immer wieder folgende Fragen:

- Wie können wir Daten cachen und wiederverwenden, die über HTTP abgerufen wurden?
- Wie machen wir Daten für mehrere Komponenten gleichzeitig verfügbar?
- Wie reagieren wir an verschiedenen Stellen auf Ereignisse, die in der Anwendung auftreten?
- Wie verwalten wir die Daten, die über die gesamte Anwendung verteilt sind?

Zustände zentralisieren

Eine häufige Lösung für all diese Herausforderungen ist die *Zentralisierung*. Liegen die Daten an einem zentralen Ort in der Anwendung vor, so können sie von überall aus genutzt und verändert werden. Diesen Schritt geht man häufig ganz selbstverständlich, indem man etwa an einer geeigneten Stelle (z. B. im `BookStoreService`) einen Cache einbaut. Doch die Idee der Zentralisierung kann man noch viel weiter gehen: Bislang waren Komponenten die »Hüter« der Daten. Jede Komponente hatte ihren eigenen Zustand und bildete eine abgeschottete Einheit zu den anderen Komponenten. Diese Idee wollen wir nun auf den Kopf stellen. Die Komponenten sollen dazu ihre bisherige Kontrolle über die Daten und die Koordination der Prozesse an eine zentrale Stelle abgeben. Die Aufgabe der Komponenten ist es dann nur noch, Daten zu lesen und Events an die zentrale Stelle zu senden. Diese Art der Zentralisierung stellt einen entscheidenden Unterschied zum bisherigen Vorgehen dar, wo alle Zustände über den gesamten Komponentenbaum hinweg verteilt waren.

Wir wollen in diesem Kapitel besprechen, wie eine solche zentrale Zustandsverwaltung (engl. *State Management*) realisiert werden kann. Dabei lernen wir das Architekturmuster *Redux* kennen und nutzen die populäre Bibliothek *Reactive Extensions for Angular (NgRx)*, um den Anwendungszustand zu verwalten und unsere Prozesse zu koordinieren.

20.1 Ein Modell für zentrales State Management

Um uns der Idee des zentralen State Managements von Redux zu nähern, wollen wir zunächst ein eigenes Modell ohne den Einsatz eines Frameworks entwickeln. Wir beginnen mit einem einfachen Beispiel, verfeinern die Implementierung schrittweise und nähern uns so der finalen Lösung an.

Objekt in einem Service

Um alle Daten und Zustände zu zentralisieren, legen wir in einem zentralen Service ein Zustandsobjekt ab. Wir definieren die Struktur dieses Objekts mit einem Interface, um von einer starken Typisierung zu profitieren. Als möglichst einfaches Beispiel dient uns eine Zahl, die man mithilfe einer Methode hochzählen kann. Unser State kann natürlich noch weitere Eigenschaften besitzen; wir haben dies mit dem Property `anotherProperty` angedeutet.

Listing 20–1
Service mit zentralem Zustand

```
export interface MyState {
  counter: number;
  anotherProperty: string;
}

@Injectable({ providedIn: 'root' })
export class StateService {
  state: MyState = {
    counter: 0,
    anotherProperty: 'foobar'
  }

  incrementCounter() {
    this.state.counter++;
  }
}
```

Unser Service hält ein Objekt mit einem initialen Zustand, das über die Methode `incrementCounter()` manipulierbar ist. Alle Komponenten können diesen Service anfordern und die Daten aus dem Objekt nutzen und verändern. Die Change Detection von Angular hilft uns dabei, automatisch bei Änderungen die Views der Komponenten zu aktualisieren.

Listing 20–2
Zentralen Zustand in der Komponente verwenden

```
@Component({ /* ... */ })
export class MyComponent {
  constructor(public service: StateService) {}
}
```

Den injizierten `StateService` können wir dann im Template nutzen, um die Daten anzuzeigen und die Methode `incrementCounter()` auszulösen:

Listing 20–3
Den Service im Template nutzen

```
<div class="counter">
  {{ service.state.counter }}
</div>
```

```
<button (click)="service.incrementCounter()">
  Increment
</button>
```

Wir haben in einem ersten Schritt unseren Zustand zentralisiert. Der Mehrwert zu einer isolierten Lösung besteht darin, dass alle Komponenten denselben Datensatz verwenden und anzeigen. Der Ort der Datenhaltung ist klar definiert und es gibt keine Datensilos bei den einzelnen Komponenten.

Subject in einem Service

Wir haben den Anwendungszustand an einer zentralen Stelle untergebracht, allerdings hat die Lösung einen Nachteil. Mit der aktuellen Architektur können wir nur über Umwege programmatisch auf Änderungen an den Daten reagieren.[2] Eine Änderung am State wird zwar jederzeit korrekt angezeigt, aber dies basiert allein auf den Mechanismen der standardmäßigen Strategie für die Change Detection.[3] Wollen wir hingegen zusätzlich eine Routine anstoßen, sobald sich Daten ändern, haben wir aktuell keine direkte Möglichkeit dazu.

Subject: Observer und Observable

Um diesen Punkt zu verbessern, ergänzen wir den Service mit einem Subject.[4] Das Subject ist ein Baustein, mit dem wir ein Event an mehrere Subscriber verteilen können. Ein Subject implementiert hierfür sowohl alle Methoden eines Observers (Daten senden) als auch die eines Observables (Daten empfangen). Wenn der Zustand geändert wird, soll das Subject diese Neuigkeit mit einem Event bekannt machen, sodass die Komponenten darauf reagieren können.

Für unser Beispiel eignet sich ein `BehaviorSubject`. Seine wichtigste Eigenschaft besteht darin, dass es den jeweils letzten Zustand speichert. Jeder neue Subscriber erhält die aktuellen Daten, ohne dass ein neues Event ausgelöst werden muss. Interessierte Komponenten können den Datenstrom also jederzeit abonnieren und auf die Ereignisse reagieren. Das `BehaviorSubject` muss mit einem Startwert initialisiert werden, der über den Konstruktor übergeben wird.

Wir setzen zunächst die Eigenschaft `state` auf privat, sodass man nun gezwungen ist, das Observable `state$` zu verwenden, anstatt di-

---

[2] Man könnte z. B. eine weitere Komponente und den Lifecycle-Hook `ngOnChanges()` einsetzen.

[3] Zur Funktionsweise und Optimierung der Change Detection in Angular haben wir im Kapitel »Wissenswertes« ab Seite 661 einen Abschnitt untergebracht.

[4] Im Kapitel zu reaktiver Programmierung mit RxJS haben wir Subjects ausführlich besprochen, siehe Seite 214.

rekt auf das Objekt zuzugreifen. Wird `incrementCounter()` aufgerufen und der State aktualisiert, so lösen wir das `BehaviorSubject` mit dem aktuellen State-Objekt aus. So werden alle Subscriber über den neuen Zustand informiert.

Listing 20–4
Zentralen Zustand mit Subject verwenden

```
@Injectable({ providedIn: 'root' })
export class StateService {
  private state: MyState = { /* ... */ }

  state$ = new BehaviorSubject<MyState>(this.state);

  incrementCounter() {
    this.state.counter++;
    this.state$.next(this.state);
  }
}
```

Unsere Komponenten können nun die Informationen aus dem Subject beziehen. Der Operator `map()` hilft uns, schon in der Komponentenklasse die richtigen Daten aus dem State-Objekt zu selektieren.

Listing 20–5
Zustand vor der Verwendung transformieren

```
@Component({ /* ... */ })
export class MyComponent {
  counter$ = this.service.state$.pipe(
    map(state => state.counter)
  );

  // ...
}
```

Im Template nutzen wir schließlich die AsyncPipe, um das Observable zu abonnieren.

Listing 20–6
Ergebnis mit der AsyncPipe anzeigen

```
<div class="counter">
  {{ counter$ | async }}
</div>

<button (click)="service.incrementCounter()">
  Increment
</button>
```

Dieser Ansatz bietet einen Mehrwert zum vorherigen Beispiel: Die Komponenten teilen sich nicht nur die Daten, sie können auch reaktiv Änderungen entgegennehmen. Zusätzlich sind wir in der Lage, bei Bedarf die Strategie der Change Detection für die Komponente zu ändern

und so in einem komplexeren Szenario gegebenenfalls die Performance zu optimieren.

Unveränderlichkeit

Unser Beispiel hat sich gut entwickelt, hat aber noch ein grundlegendes Designproblem. Wir halten unsere Daten in einem zentralen Objekt, das mit wachsender Größe der Anwendung ebenfalls größer wird. Alle Änderungen werden *direkt* an diesem Objekt durchgeführt, und wir geben es lediglich als Referenzparameter (*Call by reference*) an die Subscriber weiter. Wir stellen uns nun vor, das Objekt hätte viele weitere Eigenschaften und eine verschachtelte Datenstruktur. Die Ereignisse zum Ändern der Daten können weiterhin aus diversen Gründen ausgelöst werden. Wie können wir nun effizient herausfinden, ob das Objekt bzw. ein Teil der verschachtelten Datenstruktur verändert wurde? Die Antwort lautet: Wir können dies nicht ohne zusätzlichen Aufwand realisieren. Um eine Änderung festzustellen, ist es notwendig, das Objekt mit einer zuvor erstellten Kopie zu vergleichen. Da wir mit Referenzen arbeiten, müssen wir langwierig jede Eigenschaft der verschachtelten Datenstruktur mit dem Gegenstück aus der Kopie vergleichen.

Objekte vergleichen

Das wollen wir ändern, indem wir das Objekt *unveränderlich* (engl. *immutable*) machen. Zur Erstellung von unveränderlichen Objekten gibt es mehrere Bibliotheken, darunter das Projekt Immutable.js.[5] Für ein simples Szenario genügt auch die JavaScript-Methode `Object.freeze()`. Damit können wir ein Objekt »einfrieren« und direkte Änderungen an den Daten verhindern. Dadurch ändert sich ein grundlegender Aspekt: Da Änderungen nicht mehr direkt am bisherigen Objekt möglich sind, werden wir gezwungen, das Objekt auszutauschen. Wir erzeugen hierfür bei jeder Änderung eine Kopie des vorherigen Objekts mit einer Ausnahme: dem zu ändernden Wert. Eine Änderung festzustellen ist nun sehr einfach: Wir müssen lediglich Referenzen vergleichen. Dies ist problemlos möglich, da wir durch die Unveränderlichkeit sicher sein können, dass keine Änderung durch direkte Manipulation des Objekts möglich sein kann. Versehentliche Änderungen sind damit ebenfalls ausgeschlossen.

Kopie erzeugen

Für unseren Anwendungsfall benötigen wir allerdings gar keine echte Unveränderlichkeit! Es reicht im Prinzip schon aus, nur so zu tun, als wäre das Objekt unveränderlich, und dies konsequent beim Programmieren einzuhalten. Wir können hierfür den Spread-Operator[6] nutzen und damit alle Eigenschaften kopieren.

[5] https://ng-buch.de/a/80 – Immutable.js

[6] Den Spread-Operator und die Rest-Syntax haben wir im Kapitel zu TypeScript ab Seite 41 erklärt.

Im folgenden Listing 20–7 demonstrieren wir die Verwendung. Die Methode `incrementCounter()` nutzt den Spread-Operator, um eine Kopie des vorherigen Objekts und damit eine neue Referenz zu erzeugen. Im selben Schritt schreiben wir den neuen Wert des Zählers in die Eigenschaft `counter`.

Listing 20–7
Objekte unveränderlich behandeln

```
@Injectable({ providedIn: 'root' })
export class StateService {
  // ...

  incrementCounter() {
    this.state = {
      ...this.state,
      counter: this.state.counter + 1
    }

    this.state$.next(this.state);
  }
}
```

Wir haben durch die »Pseudo-Immutability« den Weg geebnet, um die Strategie für die Change Detection zu optimieren: Wenn ein Objekt bei einer Änderung stets eine neue Referenz erhält, so können wir in den Kindkomponenten die Strategie `OnPush`[7] einsetzen. Dies kann die Performance der Anwendung entscheidend verbessern.

Nachrichten

Entkopplung

Wir wollen einen Schritt weiter gehen und das System noch mehr entkoppeln. So wie der Service aktuell implementiert ist, muss für jede Aktion auch eine Methode existieren, die von der Komponente aufgerufen wird, z. B. `incrementCounter()`. Idealerweise kennen die Komponenten allerdings gar keine Details über die konkrete Implementierung der Zustandsverwaltung. Koppeln wir die Bausteine zu eng aneinander, so wird es mit wachsender Größe der Anwendung immer aufwendiger, grundlegende Änderungen oder Umstrukturierungen durchzuführen.

Anstatt also für jede Aktion eine Methode anzulegen, wollen wir eine Reihe von Nachrichten vereinbaren, mit denen die Anwendung Ereignisse erfassen und Aufforderungen formulieren kann. Welche Routinen als Reaktion auf eine Nachricht anzustoßen sind, das entscheidet die Zustandsverwaltung. Die Komponenten teilen lediglich mit, was in der Anwendung passiert.

[7] Auf die Change Detection und die Strategie `OnPush` gehen wir ab Seite 661 genauer ein.

Der relevante Unterschied zu einem Methodenaufruf ist die Entkopplung: Dem System steht es frei, auf eine Nachricht zu reagieren oder sie zu ignorieren. Existiert für eine bestimmte Nachricht noch keine Logik, so tritt kein Fehler auf, sondern die Nachricht wird schlichtweg nicht behandelt. Ebenso können mehrere Teile der Anwendung gleichzeitig auf Nachrichten reagieren oder auch zeitversetzt die Nachricht verarbeiten. Zeichnet man die Nachrichten auf, so bleibt durch die Historie der Nachrichten stets ersichtlich, was in welcher Reihenfolge passiert ist.

Für das Zählerbeispiel können wir beispielsweise die Nachrichten `INCREMENT`, `DECREMENT` und `RESET` vereinbaren, die von den Komponenten zum Service geschickt werden können.

Listing 20–8 *Nachricht in den Service senden*

```
@Component({ /* ... */ })
export class MyComponent {
  constructor(private service: StateService) {}

  increment() {
    this.service.dispatch('INCREMENT');
  }
}
```

Trennung von Lesen und Schreiben

Wenn wir diese Architektur genauer betrachten, fällt auf, dass wir Lesen und Schreiben für unser Zustandsobjekt vollständig voneinander getrennt haben. Die Abonnenten wissen nicht, woher die Zustandsänderungen stammen. Die Auslöser der Nachrichten wissen nicht, ob und wie der Zustand geändert wird und wer über die Änderungen informiert wird. Die Verantwortung wurde komplett an die zentrale Zustandsverwaltung übertragen, und wir haben das System stark entkoppelt.

Berechnung des Zustands auslagern

Mit der Idee von Nachrichten zum Datenaustausch und (Pseudo-)Unveränderlichkeit im Hinterkopf wollen wir die Verwaltung des Zustands erneut überdenken. Bisher haben wir das State-Objekt im Service gepflegt und bei Änderungen über das Subject ausgegeben. Der Service hat dabei zwei Verantwortlichkeiten: den zentralen State zu halten und alle Änderungen zu berechnen.

Vermeidung von Gottobjekten

Für unser kurzes Beispiel mit einem Counter ist dies kein Problem, denn wir haben nur wenige Zeilen Code. Wenn allerdings unsere Anwendung und damit die Zustandsverwaltung komplexer wird, so wächst auch der zentrale Service mit jedem Feature immer weiter an.

Bald entsteht ein »Gottobjekt« (engl. *God object*), und das müssen wir verhindern.

Die Lösung des Problems besteht darin, die Berechnung des Zustands in eine weitere unabhängige Funktion auszulagern. Wenn wir die Funktion richtig planen, so können wir die Berechnung bei zunehmender Komplexität auch in viele unabhängige Funktionen aufteilen. Weiterhin sollten die ausgelagerten Funktionen keinen eigenen Zustand besitzen (engl. *stateless*), sodass sie bei gleichen Eingangswerten stets die gleichen Ausgangswerte erzeugen. Dadurch werden die Funktionen einfacher testbar.

Zustandslose Programmierung

Über die gesamte Laufzeit der Anwendung betrachtet basiert unser Service auf einem Strom von Nachrichten, die jeweils Zustandsänderungen auslösen können. Wir besitzen die Grundlage für ein reaktives System, nun müssen wir uns diese Eigenschaft nur noch mithilfe unserer ausgelagerten Funktionen zunutze machen. Dazu entwickeln wir zunächst die Funktion, die für jede eintreffende Nachricht entscheidet, ob und wie der Zustand verändert werden soll.

Den Datenfluss können wir dabei ganz einfach halten: Die Funktion erhält als Argumente den aktuell herrschenden Zustand und die eintreffende Nachricht.

Listing 20–9
Zustand berechnen anhand einer Nachricht

```
function calculateState(state: MyState, message: string): MyState {
  switch(message) {
    case 'INCREMENT': {
      return {
        ...state,
        counter: state.counter + 1
      }
    };

    case 'DECREMENT': {
      return {
        ...state,
        counter: state.counter - 1
      };
    }

    case 'RESET': {
      return { ...state, counter: 0 };
    }

    default: return state;
  }
}
```

Der Zustand wird also durch jede eintreffende Nachricht berechnet. Wenn Änderungen durchgeführt werden sollen, so gibt die Funktion ein neues Objekt zurück, denn wir wollen den Zustand ja unveränderlich behandeln. Trifft eine unbekannte Nachricht ein, so ist keine Änderung notwendig. Wir dürfen in diesem Fall das vorherige State-Objekt zurückgeben. Unser zentraler Service kann also wie folgt angepasst werden:

Listing 20–10 *Berechnung des States auslagern*

```
@Injectable({ providedIn: 'root' })
export class StateService {
  // ...

  dispatch(message: string) {
    this.state = calculateState(this.state, message);
    this.state$.next(this.state);
  }
}
```

In diesem Schritt wurde unser System in zwei Teile aufgeteilt. Der Service hält weiterhin den State, die Berechnung wird von einer ausgelagerten Funktion durchgeführt. Durch diese Trennung bleibt der Service schlank und übersichtlich.

Deterministische Zustandsänderungen

In JavaScript existiert die Methode `Array.reduce()`. Sie hat die Aufgabe, ein Array auf einen einzigen Wert zu reduzieren, indem für jeden Wert ein Callback ausgeführt wird:

Listing 20–11 *Addition mit Array.reduce()*

```
const values = [1, 2, 3, 4];
const reducer = (previousValue, currentValue) => previousValue +
    ↪ currentValue;

// Erwartetes Ergebnis: 1 + 2 + 3 + 4 = 10
const result = values.reduce(reducer, 0);
```

Die Signatur unserer zuvor ausgelagerten Funktionen entspricht bereits den Callbacks, die auch für `Array.reduce()` verwendet werden. Unseren Zustand können wir demnach auch wie folgt berechnen:

Listing 20–12 *Nachrichten auf den Zustand reduzieren*

```
const initialState = {
  counter: 0,
  anotherProperty: 'foobar'
};
const messages = ['INCREMENT', 'DECREMENT', 'INCREMENT'];
```

```
// Erwartetes Ergebnis: { counter: 1 }
const result = messages.reduce(calculateState, initialState);
```

Mit einer solchen Reducer-Funktion und einer Liste von Nachrichten können wir demnach jeden gewünschten Zustand erzeugen. Wichtig ist dabei vor allem, dass die Reducer-Funktion »pure« ist. Sie liefert also für die gleichen Eingabewerte stets die gleiche Ausgabe. Dies ist immer dann gegeben, wenn die Funktion ausschließlich die übergebenen Parameter verwendet und keinen eigenen Zustand verwaltet.

In den vorherigen Beispielen haben wir allerdings kein Array von Nachrichten verwendet, sondern alle eingehenden Nachrichten wurden direkt an `calculateState()` weitergegeben. Wir wollen den Service nun etwas umstrukturieren: Dazu setzen wir ein Subject ein, das alle Nachrichten nacheinander in einem Datenstrom `messages$` liefert. Wir wollen erneut die Funktion `calculateState()` nutzen, um aus der Sammlung aller Nachrichten den jeweils neuen Zustand zu generieren. Dieses Mal greifen wir auf das große Toolset von RxJS zurück und verwenden den Operator `scan()`. Das ehemalige `BehaviorSubject` für den State wird von `shareReplay(1)` abgelöst, um das resultierende Observable mit allen Subscribern zu teilen und den jeweils letzten Wert an alle neuen Subscriber zu übermitteln. Um den Prozess einmalig anzustoßen, nutzen wir außerdem den Operator `startWith()` und erzeugen ein erstes Element im Strom der Nachrichten.

Listing 20–13
Nachrichten auf den Zustand reduzieren mit RxJS

```
const initialState = {
  counter: 0,
  anotherProperty: 'foobar'
};

const state$ = messages$.pipe(
  startWith('INIT'),
  scan(calculateState, initialState),
  shareReplay(1)
);
```

Das Ergebnis ist ein Observable, das für jede eintreffende Nachricht den neuen Zustand ausgibt, der von der Funktion `calculateState()` berechnet wurde. Ausgehend vom Startzustand werden also alle Nachrichten »aufsummiert« – daraus ergibt sich immer der aktuelle Zustand. Mithilfe von `scan()` müssen wir das zentrale Objekt nicht mehr selbst pflegen; dies erledigt nun RxJS für uns.

Erneut haben wir unsere Zustandsverwaltung verbessert. Der Zustand ist nun aus den gesendeten Nachrichten abgeleitet. Ist die Historie aller Nachrichten bekannt, so kann man theoretisch jeden bishe-

rigen Zustand jederzeit wieder reproduzieren, sofern unsere Reducer-Funktionen deterministisch sind.[8] Diese Eigenschaften sorgen für ein sehr einfaches und gleichzeitig robustes System. Da die Funktionen sehr simpel sind, sind sie auch sehr einfach zu testen.

Zusammenfassung aller Konzepte

Wir wollen die entwickelte Idee kurz zusammenfassen: Wir besitzen einen zentralen Service, der Nachrichten empfängt. Diese Nachrichten können von überall aus der Anwendung gesendet werden: aus Komponenten, anderen Services usw. Der Service kennt den Startzustand der Anwendung, der als ein zentrales Objekt abgelegt ist. Jede eintreffende Nachricht kann Änderungen an diesem Zustand auslösen. Der Service kennt dafür die passenden Anleitungen, wie die Nachricht zu behandeln ist und welche Änderungen am Zustand dadurch ausgelöst werden. Wird ein neuer Zustand erzeugt, wird er an alle Subscriber über ein Observable übermittelt. Jede interessierte Instanz in der Anwendung kann also die Zustandsänderungen abonnieren. Der Lesefluss und der Schreibfluss wurden vollständig entkoppelt: Die Komponenten erhalten die Daten über ein Observable und senden Nachrichten in den Service. Der Service ist die *Single Source of Truth* und hat als einziger Teil der Anwendung die Hoheit darüber, Nachrichten und Zustandsänderungen zu verarbeiten.

Redux

Wir haben schrittweise ein robustes Modell für zentrales State Management entwickelt und dabei die Idee des Redux-Patterns kennengelernt.

20.2 Das Architekturmodell Redux

Redux ist ein populäres Pattern zur Zustandsverwaltung in Webanwendungen. Die Idee von Redux stammt ursprünglich aus der Welt des JavaScript-Frameworks React, das neben Angular eines der populärsten Entwicklungswerkzeuge für Single-Page-Anwendungen ist. Redux ist dabei zunächst eine Architekturidee, es gibt aber auch eine konkrete Implementierung in Form einer Bibliothek.

Der zentrale Bestandteil der Architektur ist ein *Store*, in dem der gesamte Anwendungszustand als eine einzige große verschachtelte Datenstruktur hinterlegt ist. Der Store ist die »Single Source of Truth«

[8] Um jeden gewünschten Zustand wieder reproduzieren zu können, müsste man die Historie aller Nachrichten speichern. Das tun wir in diesem Beispiel nicht, und auch in der praktischen Anwendung von Redux wird das Protokoll der Nachrichten nicht gespeichert.

für die Anwendung und enthält alle Zustände: vom Server heruntergeladene Daten, gesetzte Einstellungen, die aktuell geladene Route oder Infos zum angemeldeten Nutzer – alles, was sich zur Laufzeit in der Anwendung verändert und den Zustand beschreibt.

State ist immutable und read-only.

Das State-Objekt im Store hat zwei elementare Eigenschaften: Es ist *immutable* (oder wird so behandelt, als wäre es unveränderbar) und *read-only*. Wir können die Daten aus dem State nicht verändern, sondern ausschließlich lesend darauf zugreifen. Möchten wir den State »verändern«, so muss das existierende Objekt durch eine Kopie ausgetauscht werden, die die Änderungen enthält. Solche Änderungen am State werden durch Nachrichten ausgelöst, die aus der Anwendung in den Store gesendet werden. Die Grundidee dieser Architektur haben wir bereits in der Einleitung zu diesem Kapitel gemeinsam entwickelt.

Bausteine von Redux

Redux verwendet vier wesentliche Bausteine: Der *Store* ist die zentrale Einheit, die den Zustand der Anwendung verwaltet. Im Store befindet sich der *State* als ein großes verschachteltes Objekt, das die Zustände der gesamten Anwendung beinhaltet. Der State kann nur gelesen werden. Alle fachlichen Ereignisse in der Anwendung werden mit Nachrichten abgebildet – im Kontext von Redux nennt man diese Nachrichten *Actions*. Eine Action wird von der Anwendung (z. B. von den Komponenten) in den Store gesendet (engl. *dispatch*) und kann eine Zustandsänderung auslösen. Im Store werden die eingehenden Actions von *Reducern* verarbeitet. Diese Funktionen nehmen den aktuellen State und die neue Action als Grundlage und errechnen daraus den neuen State. Der Datenfluss in der Redux-Architektur ist in Abbildung 20–1 grafisch dargestellt. Hier ist klar erkennbar, dass die Daten stets in eine Richtung fließen und dass Lesen und Schreiben klar voneinander getrennt sind.

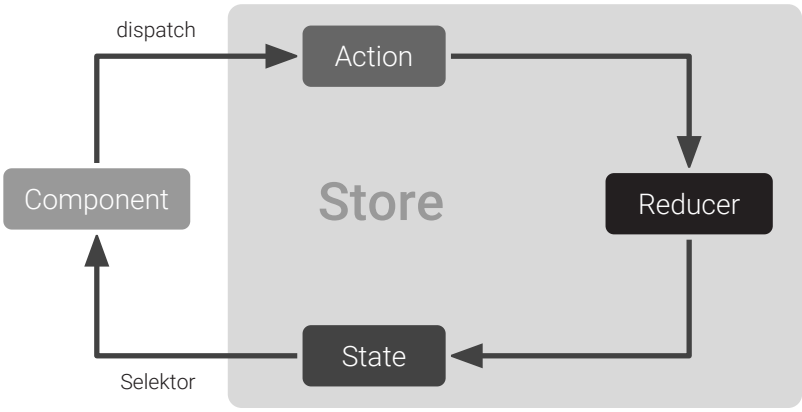


Abb. 20–1
Datenfluss in Redux

Bringt man diese Bausteine in den Kontext des einführenden Beispiels, so entspricht der zentrale Service dem Store von Redux. Die gesendeten Nachrichten entsprechen den Actions. Die Funktion `calculateState()`, die wir zur Veranschaulichung verwendet haben, ist genauso aufgebaut wie die Reducer von Redux. Der Operator `scan()` ist tatsächlich auch die technische Grundlage des Frameworks NgRx, das wir in diesem Kapitel für das State Management nutzen werden.

Redux und Angular

Die originale Implementierung von Redux stammt aus der Welt von React. Alle enthaltenen Ideen können aber problemlos auch auf die Architektur einer Angular-Anwendung übertragen werden. Es existieren verschiedene Bibliotheken, die ein zentrales State Management für Angular ermöglichen. Sie alle folgen der grundsätzlichen Idee von Redux.

- Reactive Extensions for Angular (NgRx)
- angular-redux
- NGXS
- Akita

NgRx ist das bekannteste Projekt aus dieser Kategorie. Die Bibliothek wurde von Mitgliedern des Angular-Teams aktiv mitentwickelt und gilt als De-facto-Standard für zentrales State Management mit Angular. Angular-Redux nutzt die originale Redux-Implementierung von React und stellt Wrapper zur Verfügung, um sich nahtlos in Angular zu integrieren.[9] Es lohnt sich außerdem, einen Blick auf die Community-Projekte NGXS und Akita zu werfen.

Welche der Bibliotheken Sie für die Zustandsverwaltung einsetzen sollten, hängt von den konkreten Anforderungen und auch von persönlichen Präferenzen ab. Sie sollten alle Projekte vergleichen und Ihren Favoriten nach Kriterien wie Codestruktur und Features auswählen. Dazu möchten wir Ihnen einen Blogartikel empfehlen, in dem NgRx, NGXS, Akita und eine eigene Lösung mit RxJS gegenübergestellt werden.[10]

[9] Angular-Redux haben wir in der ersten Ausgabe des Buchs vorgestellt. Mit dieser Ausgabe setzen wir jedoch auf NgRx.

[10] https://ng-buch.de/a/81 – Ordina JWorks Tech Blog: NGRX vs. NGXS vs. Akita vs. RxJS: Fight!

20.3 Redux mit NgRx

Reactive Extensions for Angular (NgRx) ist eine der populärsten Implementierungen für State Management mit Angular. Durch die gezielte Ausrichtung auf Angular fügt sich der Code gut in die Strukturen und Lebenszyklen einer Angular-Anwendung ein. Die Bibliothek setzt stark auf die Möglichkeiten der reaktiven Programmierung mit RxJS, ist also an vielen Stellen von Observables und Datenströmen geprägt. Die große Community und eine Reihe von verwandten Projekten machen NgRx zum wohl bekanntesten Werkzeug für Zustandsverwaltung mit Angular.

Wir wollen in diesem Kapitel die Struktur und die Bausteine in der Welt von NgRx genau besprechen. Außerdem wollen wir den BookMonkey mit NgRx umsetzen, um so alle Bausteine auch praktisch zu üben.

20.3.1 Projekt vorbereiten

Als Grundlage für diese Übung verwenden wir das Beispielprojekt BookMonkey in der finalen Version aus Iteration 7. Möchten Sie mitentwickeln, so können Sie Ihr bestehendes BookMonkey-Projekt verwenden oder neu starten und den Code über GitHub herunterladen:

https://ngbuch.de/bm3-it7-i18n

20.3.2 Store einrichten

Im Projektverzeichnis müssen wir zunächst alle Abhängigkeiten installieren, die wir für die Arbeit mit NgRx benötigen. NgRx verfügt über eigene Schematics zur Einrichtung in einem bestehenden Projekt der Angular CLI. Die folgenden Befehle integrieren einen vorbereiteten Store:

```
$ ng add @ngrx/store
$ ng add @ngrx/store-devtools
$ ng add @ngrx/effects
```

Später wollen wir einen zusätzlichen Baustein kennenlernen, der im originalen Redux nicht vorgesehen ist und der spezifisch für NgRx ist: Effects auf Basis von `@ngrx/effects`. Deshalb haben wir das nötige Paket

in diesem Schritt gleich mit eingefügt. Die Store DevTools sind hilfreich zum Debugging der Anwendung – wir werden im Powertipp ab Seite 597 genauer darauf eingehen, um den Lesefluss in diesem Kapitel nicht zu unterbrechen.

20.3.3 Schematics nutzen

Um nach der Einrichtung weitere Bausteine von NgRx mithilfe der Angular CLI anzulegen, können wir das Paket `@ngrx/schematics` nutzen. Es erweitert die Fähigkeiten der Angular CLI, sodass wir unsere Actions, Reducer und Effects bequem mithilfe von `ng generate` anlegen können. Auch diese Abhängigkeit wird mittels `ng add` installiert.

```
$ ng add @ngrx/schematics --defaultCollection
```

Default Collection festlegen

Mit dem Parameter `--defaultCollection` werden die Schematics von NgRx als Standardkollektion für unser Projekt festgelegt. Das bedeutet, dass jeder Aufruf von `ng generate` auf die Skripte in diesem Paket zurückgreift. So können wir bequem einen Befehl wie `ng generate action` verwenden, ohne die Zielkollektion gesondert angeben zu müssen. Da die NgRx-Schematics von den normalen Schematics für ein Angular-Projekt abgeleitet sind, funktionieren die bereits bekannten Bauanleitungen wie `ng generate component` weiterhin. Die Default Collection wird mit einem Eintrag in der Datei `angular.json` festgelegt, den Sie jederzeit wieder löschen oder ändern können, falls Sie eine andere Kollektion nutzen möchten.

20.3.4 Grundstruktur des Stores

Die ausgeführten Befehle haben bereits alles Nötige eingerichtet, sodass wir sofort mit der Implementierung beginnen können. Vorher wollen wir jedoch einen Blick auf die neu angelegten Dateien und Imports werfen.

Neu hinzugekommen ist die Datei `reducers/index.ts`. Hier befindet sich ein Interface mit dem Namen `State`, das das Kernstück unseres Anwendungszustands ist: Dieses Interface definiert, wie der gesamte State strukturiert ist, und beschreibt damit den Root-State der Anwendung. Direkt darunter befindet sich in der Variable `reducers` eine sogenannte `ActionReducerMap`. Hier ist festgelegt, welcher Reducer für welchen Teil des States verantwortlich ist. Alle Reducer der Anwendung werden also hier zusammengefasst. Zur Erinnerung: Ein Reducer verwaltet den State, indem er anhand einer Action einen neuen Zustand erzeugt. Wir werden gleich genauer darauf eingehen.

22 NativeScript: mobile Anwendungen entwickeln

»NativeScript brings together the convenience of web development with the capabilities and performance of the native mobile world.«

Stanimira Vlaeva
(Google Developer Expert und Mitglied im NativeScript Core Team)

Wir haben Angular bisher stets dafür eingesetzt, Webanwendungen zu entwickeln, die in einem Webbrowser laufen. Der Browser ist allerdings nur eine von vielen Plattformen, in denen Angular arbeiten kann.

Native Mobile-Apps mit Angular

NativeScript ist eine Toolsammlung, mit der wir native Apps für Android und iOS entwickeln können. Das HTML-Markup wird allerdings nicht von einem Webbrowser gerendert, sondern in native View-Elemente umgesetzt.

In diesem Kapitel stellen wir die Konzepte von NativeScript vor. Wir werden dabei am Beispiel erfahren, wie sich Angular auch in andere Umgebungen als in den Webbrowser nahtlos integriert. Außerdem wollen wir den BookMonkey auf NativeScript portieren.

22.1 Mobile Apps entwickeln

Die Anforderungen an moderne Apps sind unter anderem eine ansprechende Ästhetik, ein plattformspezifisches Nutzererlebnis und natürlich bestmögliche Performance. Normalerweise werden hierzu eigenständige Apps für die beiden großen mobilen Betriebssysteme erstellt. Doch parallele Entwicklungen erzeugen gleichzeitig erhöhte Kosten.

Eigenständige Apps für jede Plattform

Eine Antwort darauf sind hybride Apps auf Basis von HTML und JavaScript. Ein bekannter Vertreter dieser Kategorie ist das Framework *Ionic*.[1] Auch Ionic können wir direkt mit Angular nutzen. Die Entwicklung einer hybriden App bringt jedoch ein paar technische Beschränkungen mit sich. Durch NativeScript ist es möglich, direkt mit JavaScript native Apps zu entwickeln. Diese Apps sind nicht mehr von

Hybride Apps mit HTML

Native Apps ohne HTML

[1] https://ng-buch.de/a/88 – Ionic: Cross-Platform Mobile App Development

Lösungen unterscheidbar, die klassisch auf Basis von Objective-C bzw. Swift oder Java entwickelt worden sind. NativeScript bietet eine vergleichbare Performance wie eine native App und verwendet die normalen Bedienelemente des jeweiligen mobilen Betriebssystems.

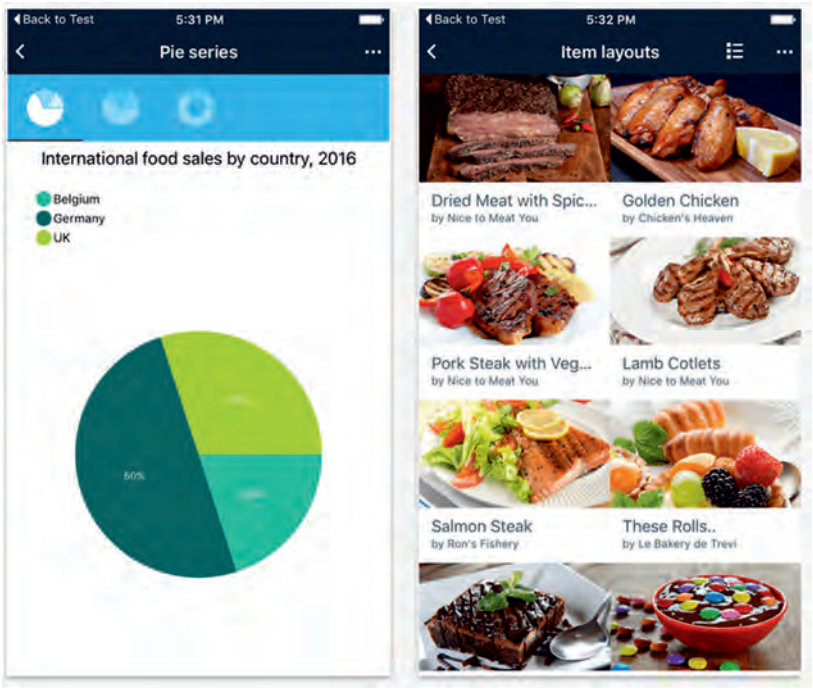


Abb. 22–1 *Zwei NativeScript-Screens unter iOS*

22.2 Was ist NativeScript?

Framework für mobile Apps

NativeScript[2], auch häufig als `{N}` abgekürzt, ist ein Open-Source-Framework zur Entwicklung von mobilen Apps. Neben dem reinen JavaScript wird auch TypeScript direkt unterstützt, was eine gute Grundlage für die Arbeit mit Angular ist. Aktuell stehen als Zielplattform sowohl Android als auch iOS zur Verfügung.

Native Apps mit JavaScript

Auf den ersten flüchtigen Blick scheint das Framework eine weitere Variante des hybriden Ansatzes auf Grundlage von HTML zu sein. Doch dem ist nicht so: NativeScript reiht sich in eine völlig neue Disziplin ein. Hier geht es darum, JavaScript als vollwertige Programmiersprache für Apps zu etablieren. Weitere Frameworks, die native Apps auf Grundlage von JavaScript ermöglichen, sind *React Native* von Facebook und *Appcelerator Titanium*. Bei allen drei Lösungen fällt der Umweg über HTML und den DOM weg. Die Frameworks ermöglichen

[2] https://ng-buch.de/a/89 – NativeScript

die direkte Verwendung von nativen UI-Elementen aus der JavaScript-Umgebung heraus. Bei NativeScript für Android ist diese Umgebung Googles V8 Engine.[3] Unter iOS kommt JavaScriptCore[4] zum Einsatz.

22.3 Warum NativeScript?

Die technische Grundlage mag zwar spannend sein, doch im Projektalltag zählen praktische Gründe. Eine Reihe von Gegebenheiten spricht für den Einsatz von NativeScript.

Wiederverwendung von bestehenden Skills

Das Erlernen einer neuen Programmiersprache zum Zwecke der App-Entwicklung ist anstrengend und aufwendig. Der Erwerb von Grundlagen einer Programmiersprache ist dabei noch das kleinere Problem. Der eigentliche Aufwand liegt im Detail. Es ist ein mühsamer und intensiver Prozess, bis ein Neueinsteiger tatsächlich alle Aspekte einer Programmierwelt kennt und sicher beherrschen kann. Während dieser Einarbeitung stehen die Entwickler natürlich nicht mehr mit dem gewohnten Potenzial und der üblichen Kapazität zur Verfügung.

Erlernen neuer Technologien kostet Zeit.

Durch die Kenntnisse um Angular steht uns bereits ein großer Teil des notwendigen Wissens zur Verfügung. Wir können ganz einfach weiter in TypeScript entwickeln und das bekannte Tooling (wie etwa Visual Studio Code) weiter verwenden.

Wiederverwendung von bestehendem Code

Durch den Einsatz der Programmiersprache TypeScript bietet es sich an, bestehende Geschäftslogik oder Bibliotheken aus dem Internet weiterzuverwenden. Mit dem Repository NPM steht ein großer Fundus von JavaScript-Bibliotheken zur Auswahl.

NPM-Pakete

Wenn wir zum Beispiel ein Datum formatieren wollen, dann können wir dafür die bekannte Bibliothek *moment* nutzen. Nach einer Installation mit `npm install moment` steht uns die Funktionalität auch in NativeScript wie üblich zur Verfügung:

```
import moment from 'moment';
const formattedTime = new moment().format('HH:mm:ss');
```

Listing 22–1
Verwendung eines NPM-Pakets

Hier ist allerdings etwas Vorsicht geboten, denn NativeScript bietet keinen DOM und kennt auch die Schnittstellen von Node.js nicht. Es funktionieren also nicht alle NPM-Pakete uneingeschränkt in NativeScript.

[3] https://ng-buch.de/a/17 – Google V8

[4] https://ng-buch.de/a/90 – JavaScriptCore

Native Bibliotheken

Neben JavaScript-Bibliotheken ist es übrigens auch möglich, bestehende native Fremdbibliotheken für Android und iOS anzusprechen. Das bedeutet, dass wir nicht in der JavaScript- bzw. NativeScript-Welt gefangen sind. Wenn es notwendig ist, können wir auch sehr plattformspezifischen Code aufrufen. Von diesem Prinzip macht auch die Komponentensammlung »NativeScript UI«[5] Gebrauch. Die bestehenden Komponentensammlungen sind hier vom Hersteller mit einem JavaScript-Wrapper vereinheitlicht worden.

Direkter Zugriff auf native APIs

Manchmal werden wir einfach nicht drum herumkommen und müssen tief in das darunterliegende Betriebssystem einsteigen. Für diese Fälle bietet NativeScript den direkten Zugriff auf native APIs aus JavaScript heraus an. Das ist ein großer Vorteil gegenüber React Native und Appcelerator, wo dies nicht so einfach möglich ist. Diesen Aspekt werden wir gleich noch einmal näher beleuchten.

Open Source

Apache License 2.0

Die Gretchenfrage in Sachen Software lässt sich bei NativeScript ohne Bauchschmerzen beantworten. Ja, das Framework ist Open Source! Es steht unter der *Apache License, Version 2.0* (ASLv2), welche die Kombination mit proprietärem Code erlaubt. Es ist problemlos möglich, einen kompletten NativeScript-Workflow mittels der offenen *NativeScript CLI*[6] aufzubauen. Zusätzlich existieren kommerzielle Angebote für komplexe Widgets und Enterprise Support.

Nahtlose Integration in Angular und die Angular CLI

NativeScript lässt sich sehr einfach mit dem Workflow der Angular CLI verknüpfen. Dazu bietet das Projekt sogenannte *Schematics* an – jene Skripte, die hinter Befehlen wie `ng generate` stecken. Wir können das Projekt von Anfang an für mehrere Plattformen ausrichten – Web, Android und iOS – und behalten stets eine einheitliche Projektstruktur. Auch wenn wir uns erst später dazu entscheiden, NativeScript in unserem Projekt zu nutzen, helfen uns die Schematics, NativeScript in unsere bestehende Struktur zu integrieren. Somit müssen wir lediglich neue Templates anlegen, können aber die gesamte Geschäftslogik unserer Anwendung übernehmen.

---

[5] https://ng-buch.de/a/91 – NativeScript UI: Professional UI Components

[6] https://ng-buch.de/a/92 – NPM: NativeScript CLI

24 Wissenswertes

»The Angular community has fundamentally changed my life, and that is not an exaggeration. I came to the community with very little frontend experience and was welcomed with open arms. This community is very special in that we see the value of pouring into the next generation and new people.«

Zackary Chapple
(Google Developer Expert und Gründer von ngAtlanta)

Dieses Buch ist vorwiegend für Einsteiger gedacht, und ganz bewusst haben wir bei der Entwicklung unserer Beispielanwendung auf bestimmte Themen verzichtet. Auf manche Dinge wollen wir dennoch eingehen, auch wenn sie in den bisherigen Kapiteln keinen Platz gefunden haben. Wir haben deshalb in diesem Abschnitt einige weiterführende Themen gesammelt, die wir Ihnen kurz vorstellen möchten.

24.1 Container und Presentational Components

Mit zunehmender Größe der Anwendung erhalten unsere Komponenten immer mehr Abhängigkeiten. Egal, ob Sie ein zentrales State Management verwenden oder mehrere einzelne Services nutzen: Viele Komponenten in der Anwendung fordern Abhängigkeiten über ihren Konstruktor an. Das erschwert inbesondere das Testing: Müssen wir Abhängigkeiten ausmocken, wird der Test komplizierter und fehleranfälliger. Auch die Austauschbarkeit ist gefährdet: Möchten wir eine Komponente ersetzen, so müssen wir darauf achten, dass auch alle Abhängigkeiten berücksichtigt werden. Nicht zuletzt erleichtert eine klare Struktur allen Entwicklern die Übersicht im Projekt. Eine solche Struktur lässt sich mit dem Konzept der Container und Presentational Components umsetzen.

Separation of Concerns

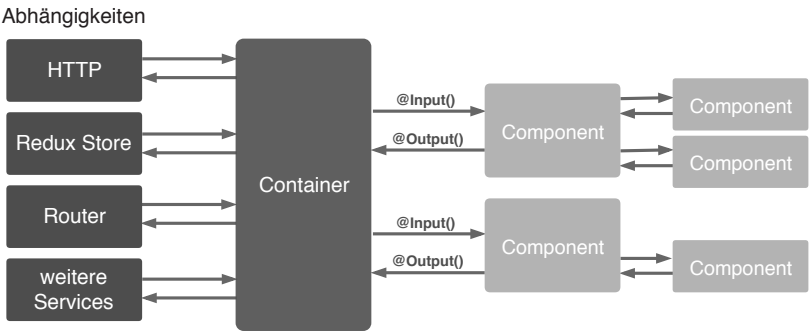
Die Grundidee besteht darin, die Zuständigkeiten der Komponenten klar aufzuteilen: Presentational Components sind ausschließlich für die Darstellung verantwortlich und kommunizieren nicht mit Services.

Container Components hingegen erledigen die Kommunikation in die Anwendung und orchestrieren die Presentational Components – besitzen aber kein eigenes umfangreiches Markup. Dadurch verringert sich die Zahl der Komponenten, die Abhängigkeiten besitzen. Gleichzeitig entstehen mehr »dumme« Komponenten, die einfach testbar, wartbar und austauschbar sind.

Smart and Dumb Components

Das Pattern ist auch unter dem Namen *Smart and Dumb Components* bekannt. Um jedoch den »unfairen« Begriff *dumb* zu vermeiden (schließlich ist es ziemlich schlau, eine klare Architektur zu entwickeln), wollen wir die neutralere Bezeichnung *Container und Presentational Components* verwenden.

Abb. 24–1 *Container und Presentational Components*



Presentational Components

Darstellung durch eigenes Markup

Presentational Components sind für die Darstellung verantwortlich. Sie besitzen immer eigenes Markup im Template und definieren damit die Teile der Anwendung, in denen sich tatsächlich sichtbare Segmente der UI befinden. Die Kommunikation mit der Anwendung findet ausschließlich über Bindings statt; eine Presentational Component darf in der Regel keine Services über ihren Konstruktor injizieren. Es gibt jedoch auch Ausnahmen wie z. B. einen injizierten Service zur asynchronen Validierung von Formularen. Das bedeutet, dass eine solche Komponente alle Daten über Input-Propertys erhält. Alle ausgehenden Daten werden über Events mithilfe von Output-Propertys kommuniziert. Dazu gehören insbesondere Nutzeraktionen, die in der Komponente erfasst werden, z. B. Button-Klicks oder Absenden eines Formulars. Unter einer solchen Komponente können im Baum auch weitere Presentational Components angeordnet werden.

Keine Services, sondern Kommunikation mit Bindings

Durch diese klar definierten Kommunikationswege wird die Zuständigkeit der Komponenten begrenzt: Presentational Components erhalten Daten zur Anzeige und geben erfasste Daten aus. Sie wissen allerdings nicht, woher die eingehenden Daten stammen oder wie die ausge-

henden Daten verarbeitet werden. Dadurch sind solche Komponenten einfach wiederverwendbar und austauschbar. Stellen Sie sich am besten vor, sie entwickeln Komponenten für eine Drittanbieter-Bibliothek. Niemand möchte eine externe Komponente einsetzen, die sehr anwendungsspezifische Logik enthält.

Presentational Components im BookMonkey

Wir haben im BookMonkey bereits intuitiv Presentational Components entwickelt: Die `BookListItemComponent` aus Iteration I erhält lediglich ein einzelnes Buch zur Anzeige und besitzt keine weiteren Abhängigkeiten. Auch die `BookFormComponent` aus Iteration IV ist praktisch eine Presentational Component: Sie nutzt zwar den `BookExistsValidatorService` zur Validierung der Formularwerte, allerdings ist diese Ausnahme legitim. Hier ist eine hohe Wiederverwendbarkeit zu erkennen: Die Komponente wird zum Anlegen und Bearbeiten eines Buchs eingesetzt, sie hat aber keine Kenntnis darüber, in welchem Kontext sie verwendet wird oder wie die Daten verarbeitet werden.

Presentational Components können übrigens in den meisten Fällen die Change-Detection-Strategie `OnPush` verwenden, denn sie kommunizieren nur über Bindings mit dem Rest der Anwendung. Die Change Detection besprechen wir ab Seite 661 im Detail und betrachten dort auch die `OnPush`-Strategie.

Container Components

Datenhaltung und Servicekommunikation

Die Aufgaben der Container Components sind die Datenhaltung und Kommunikation mit der gesamten Anwendung. Sie verfügen über Abhängigkeiten zu allen benötigten Services, etwa dem Redux Store oder dem Router. Außerdem besitzen Container kein eigenes umfangreiches Markup: Ihre einzige Aufgabe ist es, Presentational Components zu orchestrieren. Deshalb finden sich im Template keine anderen Elemente als die Host-Elemente der Kindkomponenten und ggf. einzelne `<div>`-Container zur Strukturierung.

Orchestrierung

Container müssen alle Daten für die eingebundenen Presentational Components bereitstellen und mithilfe von Property Bindings im Komponentenbaum nach unten durchreichen. Events von den Kindkomponenten müssen erfasst und verarbeitet werden.

Container sind meist sehr spezifisch für eine Seite oder einen funktionalen Teilbereich einer Seite. An oberster Stelle in der Hierarchie der Komponenten steht also immer ein Container, der die Datenhaltung organisiert und mit den Kindkomponenten kommuniziert. Beim Routing sollten Sie ausschließlich Container Components verwenden, denn durch eine geroutete Komponente wird eine Seite und damit auch die höchste Ebene der Hierarchie definiert. Dies ist immer eine sehr spezifische Angelegenheit.

Intermediäre Container

In manchen Veröffentlichungen wird vorgeschlagen, dass in einer Hierarchie von Presentational Components auch weitere Container eingefügt werden können. Das ist dann sinnvoll, wenn die Struktur zu komplex ist, um alle Daten durch den gesamten Baum zu kommunizieren. Durch solche »Zwischencontainer« lassen sich die Daten für einen Ast des Baums separat verwalten, sodass der übergeordnete Container nicht mit dieser Aufgabe betraut wird.

Observables und die AsyncPipe

Die Aufgabe von Containern ist es auch, Observables mithilfe der AsyncPipe aufzulösen und die Daten synchron an die Kindkomponenten weiterzugeben. Wir sollten im Idealfall keine Observables direkt an Property Bindings übergeben. So hat die Kindkomponente keine Kenntnis über die technische Umsetzung des Datenstroms und ist dadurch besonders gut wiederverwendbar.

Listing 24–1 *Observable auflösen im Container*

```
<my-presentational [data]="data$ | async">
</my-presentational>
```

Gegenüberstellung

In Tabelle 24–1 sind die wichtigsten Charakteristiken der beiden Komponentenarten gegenübergestellt. Elementar ist es, dass Presentational Components keine Abhängigkeiten zu Services besitzen. Außerdem dürfen Container Components nur die Orchestrierung durchführen und keine View mit gestalterischen Inhalten besitzen.

Ordnerstruktur

Es ist gute Praxis, die beiden Komponentenarten auch in der Ordnerstruktur voneinander zu trennen. Dazu eignen sich zwei Ordner mit der Bezeichnung `containers` und `components`.

Wir haben das Konzept der Container und Presentational Components im BookMonkey nicht konsequent umgesetzt, um die Komplexität der Baumstruktur nicht zu erhöhen. Sie sollten in der Praxis fallweise entscheiden, ob Sie die Trennung durchführen oder nicht. Hilfreich sind dazu Konventionen im Team, sodass alle Entwickler einen einheitlichen Stil verfolgen.

Tab. 24–1 *Gegenüberstellung Container und Presentational Components*

	Container	Presentational
Wiederverwendbarkeit	⋆	⋆⋆⋆
Austauschbarkeit	⋆	⋆⋆⋆
Abhängigkeit zu Services	⋆⋆⋆	keine
Komplexität	⋆⋆	⋆
Markup	⋆	⋆⋆⋆

Index

S