3. Auflage

Ferdinand Malcher · Johannes Hoppe · Danny Koppenhagen

Angular

Grundlagen, fortgeschrittene Themen und Best Practices

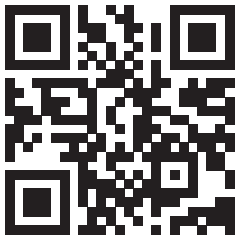
inkl. RxJS, NgRx und PWA

iX EDITION

dpunkt.verlag

Leseprobe

Angular, 3. Auflage



https://angular-buch.com

Liebe Leserin, lieber Leser,

das Angular-Ökosystem wird kontinuierlich verbessert. Bitte haben Sie Verständnis dafür, dass sich seit dem Druck dieses Buchs unter Umständen Schnittstellen und Aspekte von Angular weiterentwickelt haben können. Die GitHub-Repositorys mit den Codebeispielen werden wir bei Bedarf entsprechend aktualisieren.

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 wünschen Ihnen viel Spaß mit Angular!

Alles Gute
Ferdinand, Johannes und Danny

Ferdinand Malcher ist Google Developer Expert (GDE) und arbeitet als selbständiger Entwickler, Berater und Mediengestalter mit Schwerpunkt auf Angular, RxJS und TypeScript. Gemeinsam mit Johannes Hoppe hat er die Angular.Schule gegründet und bietet Workshops und Beratung zu Angular an.

@fmalcher01

Johannes Hoppe ist Google Developer Expert (GDE) und arbeitet als selbstständiger Trainer und Berater für Angular, TypeScript und Node.js. Zusammen mit Ferdinand Malcher hat er die Angular.Schule gegründet und bietet Schulungen zu Angular an. Johannes ist Organisator des Angular Heidelberg Meetup.

@JohannesHoppe

Danny Koppenhagen arbeitet als Softwareentwickler und Berater für Enterprise-Webanwendungen. Sein Schwerpunkt liegt in der Entwicklung von nutzerzentrierten Anwendungen mit TypeScript und Angular sowie JavaScript und Vue.js. Neben der beruflichen Tätigkeit ist Danny als Autor mehrerer Open-Source-Projekte aktiv.

@d_koppenhagen

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

Ferdinand Malcher · Johannes Hoppe · Danny Koppenhagen

Angular

Grundlagen, fortgeschrittene Themen und Best Practices – inkl. RxJS, NgRx und PWA

3., 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 und Projektkoordinierung: Julia Griebel
Copy-Editing: Annette Schwarz, Ditzingen
Satz: Da-TeX Gerd Blumenstein, Leipzig, *www.da-tex.de*
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 978-3-86490-779-1
PDF 978-3-96910-081-3
ePub 978-3-96910-082-0
mobi 978-3-96910-083-7

3., aktualisierte und erweiterte Auflage 2020

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
(ehem. 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 vier 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 von Angular 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 10)
- **AngularJS** – das Angular-Framework in der **Version 1.x.x**

AngularJS, das 2010 erschien, ist zwar mittlerweile etwas in die Jahre gekommen, viele Webanwendungen setzen aber weiterhin auf das Framework. Die letzte Version 1.8.0 wurde im Juni 2020 veröffentlicht und wird ab Januar 2022 offiziell nicht mehr weiterentwickelt.[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 durchschnittlich sechs Monaten erscheint eine neue Major-Version *x*. Die Minor-Versionen *y* werden monatlich herausgegeben, nachdem eine Major-Version erschienen ist.

Semantic Versioning

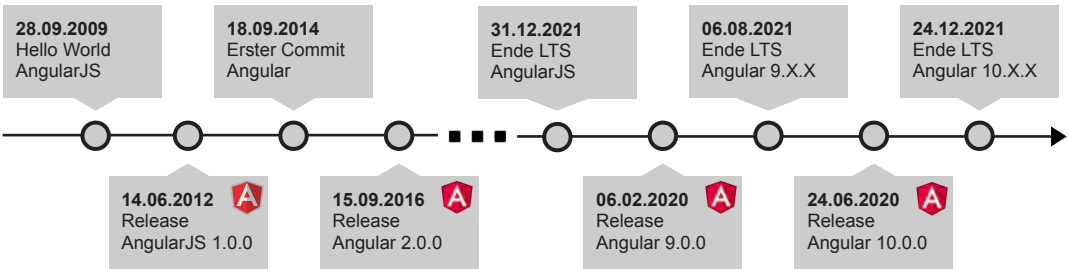


Abb. 1
Zeitleiste der Entwicklung von Angular

[1] https://ng-buch.de/b/1 – AngularJS: Version Support Status
[2] https://ng-buch.de/b/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. Entwickler ohne Vorkenntnisse in der Webentwicklung werden womöglich an manchen Stellen Hilfe zurate ziehen müssen. Wir empfehlen, in diesem Fall zunächst die grundlegenden Kenntnisse in den Bereichen HTML, JavaScript und CSS zu festigen.

Kein klassisches Nachschlagewerk

Weiterhin ist dieses Buch kein klassisches Nachschlagewerk: Wir erschließen uns die Welt von Angular praxisorientiert anhand eines Beispielprojekts. Jedes Thema wird zunächst ausführlich in der Theorie behandelt, sodass Sie die Grundlagen auch losgelöst vom Beispielprojekt nachlesen können. Dabei werden aber nicht alle Themen bis ins kleinste Detail betrachtet. Wir wollen einen soliden Einstieg in Angular bieten, *Best Practices* zeigen und Schwerpunkte bei speziellen fortgeschrittenen Themen setzen. Die meisten Aufgaben aus dem Entwicklungsalltag werden Sie also mit den vielen praktischen Beispielen souverän meistern können.

Offizielle Angular-Dokumentation

Wir hoffen, dass dieses Buch Ihr täglicher Begleiter bei der Arbeit mit Angular wird. Für Details zu den einzelnen Framework-Funktionen empfehlen wir die offizielle Dokumentation für Entwickler.[3]

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.

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.

Refactoring

Neben Storys gibt es Refactoring-Abschnitte. Dabei handelt es sich um technische Anforderungen, die die Architektur oder den Codestil der Anwendung verbessern.

Powertipps

Haben wir eine Iteration abgeschlossen, prüfen wir, ob wir unseren Entwicklungsprozess vereinfachen und beschleunigen können. In den

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

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. Dabei betrachten wir die Hintergründe und Konfiguration des Build-Prozesses und erläutern die Bereitstellung mithilfe von Docker.

Weiterführende Themen

SSR

Redux

Im fünften Teil möchten wir Ihnen mit Server-Side Rendering und der Redux-Architektur zwei Ansätze näherbringen, die über eine Standardanwendung hinausgehen. Mit *Server-Side Rendering (SSR)* machen Sie Ihre Anwendung fit für Suchmaschinen und verbessern zusätzlich die Geschwindigkeit beim initialen Start der App. 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.

Progressive Web Apps

NativeScript

Der sechste Teil dieses Buchs dreht sich um mobile Anwendungen mit Angular: Nachdem wir die Begriffe rund um das Thema *App* eingeordnet haben, besprechen wir die Ideen und Implementierung einer *Progressive Web App (PWA)* mit Angular. Abschließend betrachten wir den Einsatz von *NativeScript*, 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 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, in offenen Gruppen oder als Online-Kurs 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. Besonderer Dank geht an **Michael Kaaden** für seine unermüdlichen Anregungen, kritischen Nachfragen und seine starke Unterstützung beim Kapitel zu Docker. **Danilo Hoffmann**, **Jan Buchholz**, **Manfred Steyer** und **Jan-Niklas Wortmann** danken wir ebenso für die hilfreichen Anregungen und Korrekturvorschläge. Unser Dank geht außerdem an **Michael Hladky** für wertvollen Input zur Change Detection und zur Bibliothek RxAngular. Darüber hinaus hat uns **Nathan Walker** mit seiner Zeit und Expertise beim Kapitel zu NativeScript unterstützt.

Wir danken **Gregor Woiwode** für die Mitwirkung als Autor in der ersten Auflage. Dem Team vom dpunkt.verlag, insbesondere **René Schönfeldt**, danken wir für die persönliche Unterstützung und die guten Anregungen zum Buch. **Annette Schwarz** danken wir für das gewissenhafte Korrektorat unseres Manuskripts. Besonderer Dank gilt dem **Angular-Team** und der Community dafür, dass sie eine großartige Plattform geschaffen haben, die uns den Entwickleralltag angenehmer macht.

Viele Leser haben uns E-Mails mit persönlichem Feedback zum Buch zukommen lassen – vielen Dank für diese wertvollen Rückmeldungen.

Aus Gründen der Lesbarkeit verzichten wir in diesem Buch auf eine geschlechtsneutrale Formulierung. Wir möchten betonen, dass wir selbstverständlich durchgängig alle Personen jeden Geschlechts ansprechen.

Aktualisierungen in der dritten 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 drei 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 dritte Auflage, die Sie mit diesem Buch in Händen halten, wurde deshalb grundlegend aktualisiert und erweitert. Dabei haben wir das Feedback unserer Leser berücksichtigt, Fehler korrigiert und viele Erklärungen verständlicher formuliert.

Wir möchten Ihnen einen kurzen Überblick über die wichtigsten Neuerungen und Aktualisierungen der dritten Auflage geben. Alle Texte und Codebeispiele haben wir auf die Angular-Version 10 aktualisiert. Dabei betrachten wir auch neue Features. Schon in der zweiten Auflage haben wir umfassende Aktualisierungen vorgenommen, die wir am Ende dieses Abschnitts zusammengefasst haben.

Neue Kapitel

In der dritten Auflage sind die folgenden Kapitel neu hinzugekommen:

10.3.5 OAuth 2 und OpenID Connect (Seite 262) In der Iteration III erläutern wir die Kommunikation mit einem HTTP-Backend und betrachten Interceptoren, mit denen wir zum Beispiel Tokens in den Header einer HTTP-Nachricht einfügen können. In diesem Zusammenhang haben wir einen neuen Abschnitt hinzugefügt, in dem wir die Authentifizierung und Autorisierung mithilfe von OAuth 2 und OpenID Connect erläutern.

19 Angular-Anwendungen mit Docker bereitstellen (Seite 563)
Nachdem wir das Thema Docker in der zweiten Auflage nur in einem kurzen Abschnitt erwähnt hatten, haben wir nun ein ausführliches Kapitel neu ins Buch aufgenommen. Dort erläutern wir am praktischen

Beispiel, wie Sie eine Angular-Anwendung in einem Docker-Container bereitstellen und ausführen können. Bei diesem Kapitel hat uns unser treuer Leser *Michael Kaaden* mit Praxiserfahrung und viel Zuarbeit unterstützt – vielen Dank!

24 Progressive Web Apps (PWA) (Seite 673) Progressive Web Apps sind ein wichtiger Pfeiler für moderne Anwendungsentwicklung mit Webtechnologien. Hier spielen besonders Service Worker, Installierbarkeit, Offlinefähigkeit und Push-Benachrichtigungen eine Rolle. Da das Thema bisher in diesem Buch nicht behandelt wurde, haben wir ein neues Kapitel entwickelt. Sie lernen dabei die Grundlagen zu Progressive Web Apps und migrieren die Beispielanwendung zu einer PWA.

27 Fortgeschrittene Konzepte der Angular CLI (Seite 735) Die Angular CLI kann mehr als nur eine Anwendung in einem Projekt verwalten. In diesem neuen Kapitel werfen wir deshalb einen Blick auf die Architektur eines *Workspace*, der mehrere Anwendungen und Bibliotheken in einem gemeinsamen Repository pflegt. Zusätzlich betrachten wir hier kurz die *Schematics*, die für die Codegenerierung in der Angular CLI verantwortlich sind.

28.1 Web Components mit Angular Elements (Seite 743) Unter »Wissenswertes« sammeln wir interessante Themen, die im Verlauf des Buchs keinen Platz gefunden haben. Hier haben wir einen neuen Abschnitt zu Angular Elements hinzugefügt. Sie lernen, wie Sie Angular-Komponenten als Web Components verpacken, um sie auch in anderen Webanwendungen einzusetzen.

Stark überarbeitete und erweiterte Kapitel

10.2 Reaktive Programmierung mit RxJS (Seite 206) Das Kapitel zu RxJS haben wir um einige wichtige Details ergänzt: So wurde die Erläuterung zu Higher-Order Observables überarbeitet und mit Marble-Diagrammen illustriert, wir haben den Unterschied zwischen Observer und Subscriber stärker herausgestellt und viele Erläuterungen vereinfacht.

15.1 i18n: mehrere Sprachen und Kulturen anbieten (Seite 449) Das Thema Internationalisierung wurde mit Angular 9.0 neu aufgerollt und verfügt nun über erweiterte Funktionen, z. B. Übersetzungen im TypeScript-Code. Wir nutzen in diesem Kapitel jetzt das neue Paket `@angular/localize`, um Übersetzungen zu rendern.

18 Build und Deployment mit der Angular CLI (Seite 539) Das Kapitel zum Deployment haben wir neu strukturiert. Hier wird nun die Build-Konfiguration in der `angular.json` detailliert erläutert. Mit dem Release des neuen Ivy-Compilers ist auch das Thema JIT mehr in den Hintergrund gerückt. Außerdem haben wir einen neuen Abschnitt zum Befehl `ng deploy` hinzugefügt.

20 Server-Side Rendering mit Angular Universal (Seite 587) Der Workflow für Server-Side Rendering wurde mit Angular 9.0 stark vereinfacht. Wir haben das Kapitel zu Angular Universal aktualisiert und erweitert: Dabei gehen wir auf den neuen Builder für statisches Pre-Rendering ein, geben Tipps für den Praxiseinsatz und betrachten das Community-Projekt *Scully*.

21 State Management mit Redux und NgRx (Seite 607) Das Framework NgRx wird stetig weiterentwickelt, und so haben wir das Kapitel zum State Management grundlegend aktualisiert. Wir setzen nun durchgehend auf die neuen Creator Functions und haben viele Erläuterungen ausführlicher und verständlicher gestaltet. Außerdem gehen wir auf das neue Paket `@ngrx/component` ein und werfen einen kurzen Blick auf das Community-Projekt *RxAngular*.

Sonstiges

Fehler gefunden?

Neben den genannten Kapiteln haben wir alle Texte im Buch erneut kritisch überarbeitet. An vielen Stellen haben wir Formulierungen angepasst, Details ergänzt und Fehler korrigiert. Wenn Sie weitere Fehler finden oder Anregungen zum Buch haben, so schreiben Sie uns bitte! Wir werden uns Ihr Feedback in der nächsten Auflage zu Herzen nehmen.

Für die einzelnen Iterationsschritte aus dem Beispielprojekt bieten wir eine Differenzansicht an. So können Sie die Änderungen am Code zwischen den einzelnen Kapiteln besser nachvollziehen. Wir gehen darauf auf Seite 53 genauer ein.

Zu guter Letzt haben wir an ausgewählten Stellen in diesem Buch Zitate von 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.

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 und Sprachelemente von TypeScript erläutern, sodass es Ihnen 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 erkannt und vermieden 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 wurde jedoch eine neue Namenskonvention eingeführt. Da fortan jährlich eine neue Version erscheinen soll, wurde die Bezeichnung vom schon vielerorts 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*.

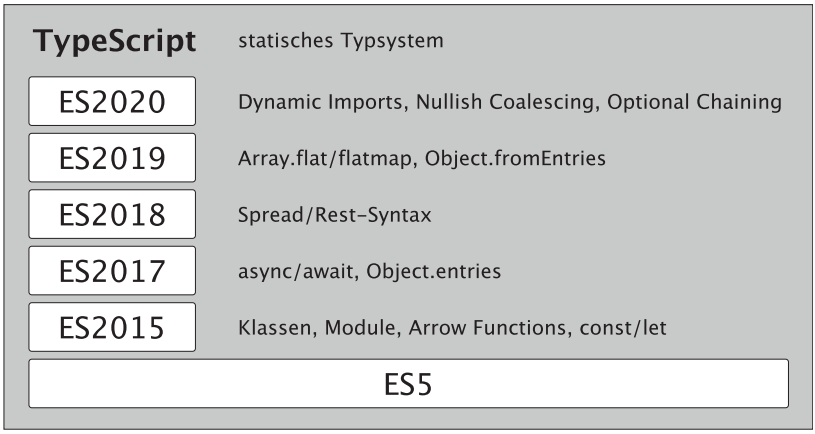


Abb. 4–1
TypeScript und ECMAScript

```
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.

TypeScript und Angular

Der empfohlene Editor Visual Studio Code unterstützt TypeScript nativ und ohne zusätzliche Plug-ins. In einem Angular-Projekt ist der TypeScript-Compiler außerdem schon vollständig konfiguriert, sodass wir sofort mit der Entwicklung beginnen können.

[1] https://ng-buch.de/b/25 – TypeScript

[2] https://ng-buch.de/b/26 – ECMAScript compatibility table

Teil III

BookMonkey 4:
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. Angular nutzt Decorators, um den Klassen eine Bedeutung zuzuordnen, z. B. `@Component()`. Im Kapitel zu TypeScript auf Seite 47 gehen wir auf Decorators ein.

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.

deren einsetzen usw. Diese Praxis schauen wir uns im nächsten Kapitel ab Seite 102 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 380 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 695) 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.

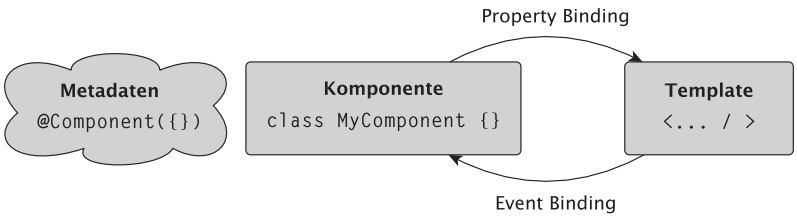


Abb. 6–1
Komponente, Template und Bindings im Zusammenspiel

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 eindeutigen 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 70.

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!

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

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

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

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

6.1.3 Template-Syntax

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

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

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

{{ Interpolation }}

Ein zentraler Bestandteil der Template-Syntax ist die Interpolation. Hinter diesem etwas sperrigen Begriff verbirgt sich die Möglichkeit, Daten mit zwei geschweiften Klammern in ein Template einzubinden.

Template-Ausdruck

Zwischen den Klammern wird ein sogenannter *Template-Ausdruck* angegeben. Dieser Ausdruck bezieht sich immer direkt auf die zugehörige Komponentenklasse. Im einfachsten Fall ist ein solcher Ausdruck also ein Name eines Propertys aus der Klasse. Ausdrücke können aber auch komplexer sein und z. B. Arithmetik enthalten. Vor der Ausgabe werden die Ausdrücke ausgewertet, und der Rückgabewert wird schließlich im Template angezeigt. Besonders interessant ist dabei: Ändern sich die Daten im Hintergrund, wird die Anzeige stets automatisch

Die Daten werden bei der Interpolation automatisch aktualisiert.

aktualisiert! Wir müssen uns also nicht darum kümmern, die View aktuell zu halten, sondern nur die Daten in der Komponentenklasse ändern.

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

Die Werte `null` und `undefined` werden übrigens immer als leerer String ausgegeben.

Der Safe-Navigation-Operator ?

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

An dieser Stelle kann der *Safe-Navigation-Operator* helfen: Er überprüft, ob das angefragte Objekt vorhanden ist oder nicht. Falls nicht, wird der Ausdruck zum Binden der Daten gar nicht erst ausgewertet.

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

Existiert das Objekt `person`, werden die Hobbys ausgegeben – es gibt aber keinen Fehler, wenn das Objekt `undefined` ist.

Operator sparsam verwenden

Aber Achtung: Bitte benutzen Sie diesen Operator sparsam. Eine bessere Praxis ist es, fehlende Objekteigenschaften mit der Strukturdirektive `ngIf` abzufangen (siehe Seite 84) und den kompletten Container erst dann anzuzeigen, wenn alle Daten vollständig sind. Ein weiterer Ansatz zur Lösung ist, die Propertys immer mit Standardwerten zu belegen. Hat ein Property immer einen Wert, muss auch nicht gegen `undefined` geprüft werden.

[Property Bindings]

Mit Property Bindings werden Daten von außen an ein DOM-Element übermittelt. Wir notieren ein Property Binding mit eckigen Klammern. Darin wird der Name des Propertys auf dem DOM-Element angegeben, das wir beschreiben wollen. Das klingt zunächst umständlich, wir werden uns aber noch ausführlich mit Property Bindings beschäftigen.

Im rechten Teil des Bindings wird ein Template-Ausdruck angegeben, also wie bei der Interpolation z. B. ein Property der aktuellen

Property Bindings werden automatisch aktualisiert.

Komponente oder ein Literal. Ist das Element ein Host-Element einer Komponente, können wir die Daten innerhalb dieser Komponente auslesen und verarbeiten. Property Bindings werden ebenfalls automatisch aktualisiert, wenn sich die Daten ändern. Wir müssen uns also nicht darum kümmern, Änderungen an den Daten manuell mitzuteilen.

Das folgende Beispiel zeigt Property Bindings im praktischen Einsatz. Wir setzen damit das Property `href` des Anker-Elements auf den Wert der Komponenteneigenschaft `myUrl`. Das zweite Beispiel setzt das Property `myProp` der Komponente `MyComponent` mit dem Wert des Propertys `foo` aus der aktuellen Komponente.

```
<a [href]="myUrl">My Link</a>
<my-component [myProp]="foo">
```

Property Bindings werden im Abschnitt ab Seite 102 ausführlich behandelt.

Eselsbrücke

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

(Event Bindings)

Gegenstück zu Property Bindings

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

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

Wie wir Events in Komponenten auslösen und welche eingebauten Events abonniert werden können, schauen wir uns ausführlich ab Seite 114 an.

Eselsbrücke

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

- Alle Methoden geben ein Observable zurück, das wir mit `subscribe()` abonnieren müssen, um die Daten zu erhalten.
- Die Aufrufe des `HttpClient` sollten in einem Service untergebracht werden. Das Observable wird allerdings nicht im Service aufgelöst, sondern bis zur Komponente durchgereicht.
- Requests mit dem `HttpClient` können mit Optionen gesteuert werden. Damit können wir z. B. Headerfelder und Query-Parameter setzen.

Demo und Quelltext:
https://ng-buch.de/bm4-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 (ein Ereignis findet statt), Sie reagieren darauf und drücken die Schlummertaste. Nach 10 Minuten klingelt der Wecker wieder, und Sie stehen auf.[3] 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 in einem Gruppenchat stammen: Einzelne Menschen haben Nachrichten erzeugt, die bei Ihnen in einem großen Datenstrom zusammengeführt wurden. Sie können die Nachrichten in der Reihenfolge lesen, wie sie eingetroffen sind.

Frühstück kaufen

Nach dem Aussteigen holen Sie sich in der Bäckerei 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 Brot. Was ist passiert? Ein Strom von eingehenden Kunden, die Geld besitzen, wurde umge-

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

wandelt in einen Strom von ausgehenden Kunden, die nun Brot 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 Backwaren kauft. Manche Ereignisse treten nur für uns und als Folge anderer Aktionen auf: Das Brot 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 passend 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 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 eine Funktion entwerfen, die über die Zeit nacheinander mehrere Werte ausgeben kann. Jeder, der an den

Werten interessiert ist, kann die Funktion aufrufen und 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 `producer()`, die wir im weiteren Verlauf auch 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 *Subscriber*. Im Körper der Producer-Funktion führen wir nun beliebige Aktionen aus, so wie es eben für eine Funktion üblich ist. Immer wenn im Funktionsablauf etwas passiert, rufen wir eins der drei Callbacks aus dem Subscriber 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 producer(subscriber) {
  setTimeout(() => {
    subscriber.next(1);
  }, 1000);

  subscriber.next(2);

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

Damit in unserem Programm auch tatsächlich etwas passiert, rufen wir die Producer-Funktion `producer()` auf und übergeben als Argument ein Objekt mit den drei Callbacks `next`, `error` und `complete`. In der Terminologie von RxJS heißt dieses Objekt *Observer*, also jemand, der den Datenstrom beobachtet.

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')
};

// Funktion aufrufen
producer(myObserver);
```

In diesem Beispiel sind *Observer* und *Subscriber* übrigens gleichbedeutend: Der Observer, den wir an die Funktion `producer()` übergeben, ist innerhalb der Funktion als Argument `subscriber` verfügbar. 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 `producer()` aufgerufen wird, startet das Routine. 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: Die Grundidee dieses Patterns ist nichts anderes als eine Funktion, die Werte an die übergebenen Callbacks ausgibt. Natürlich ist das »echte« Observable aus RxJS noch viel mehr als das. Diesen Aufbau betrachten wir in den nächsten Abschnitten noch genauer – und wir werden die hier entwickelte Producer-Funktion in dem Zusammenhang wiedersehen.

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.

Observable: Wrapper um eine Producer-Funktion

Die Observable-Implementierung von RxJS folgt grundsätzlich der Idee, die wir im letzten Abschnitt an unserem Funktionsbeispiel entwickelt haben: Das Observable ist ein Wrapper um eine Producer-Funktion. Um den Datenstrom zu abonnieren, übergeben wir einen Observer mit drei Callbacks. Damit nun Daten geliefert werden können, wird intern die Producer-Funktion ausgeführt. Der Producer ruft die Callbacks aus dem Observer auf, sobald etwas passiert.

RxJS bringt für sein Observable einen wohldefinierten Rahmen mit und befolgt einige Regeln. Dazu gehören unter anderem folgende Punkte:

- 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.
- Das Observable wandelt intern den Observer in einen Subscriber um – mehr dazu im Kasten auf Seite 213.

Wir werden den Aufbau und die Funktionsweise eines solchen Observables in den folgenden Abschnitten genauer betrachten. Behalten Sie da-

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

bei das Funktionsbeispiel im Hinterkopf, denn Sie werden einige Dinge wiedererkennen.

10.2.4 Observables abonnieren

Wir wollen uns zunächst anschauen, wie wir die Daten aus einem existierenden Observable erhalten können. Dazu 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);
```

Übrigens müssen Sie nicht immer alle drei Methoden im Observer angeben. Interessieren wir uns zum Beispiel nicht für den Fehlerfall, so reicht es, wenn der Observer lediglich `next` und `complete` beinhaltet.

Neben dieser etwas aufwendigen Schreibweise gibt es noch einen anderen Weg. Anstatt ein Objekt mit drei Methoden zu notieren, können wir die drei Callbacks auch einzeln nacheinander als Argumente von `subscribe()` angeben. Sind wir nur an den regulären Werten aus dem Observable interessiert, so reicht es sogar aus, wenn wir lediglich 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')
);
```

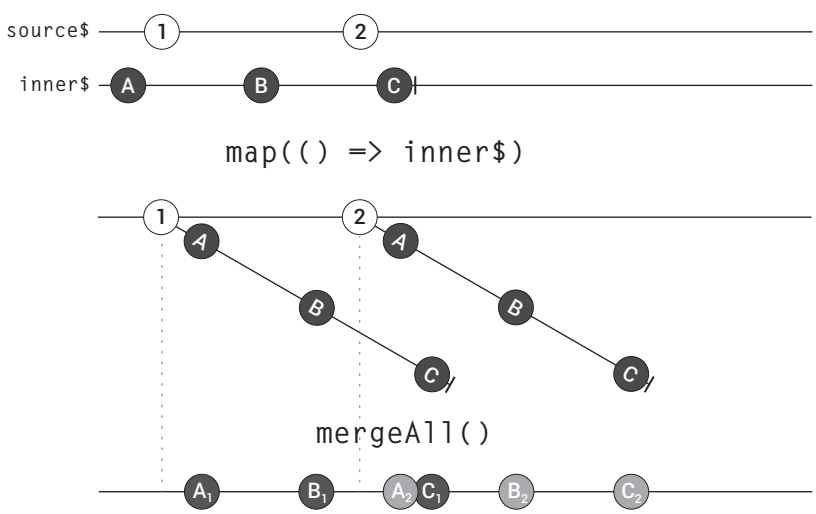


Abb. 10–9
Flattening mit mergeAll()

Der Operator `mergeMap()` kombiniert die Funktionalitäten von `map()` und `mergeAll()`:

- Er mappt die Werte eines Quell-Observables (`source$`) auf ein anderes Observable (`inner$`), bildet also das `map()` ab,
- erstellt Subscriptions auf die inneren Observables (`inner$`) und
- führt die empfangenen Daten zurück in den Hauptdatenstrom, in der Reihenfolge ihres Eintreffens.

In Bezug auf unser Beispiel bedeutet das: Für jedes Signal aus dem Trigger wird ein HTTP-Request ausgeführt. Als Ausgabe erhalten wir ein Observable, das die Ergebnisse aller dieser HTTP-Requests beinhaltet. Wichtig dabei ist, dass die Ergebnisse so ausgegeben werden, wie sie eintreffen. Braucht eine Antwort länger als eine andere, so kann sich die Reihenfolge ändern.

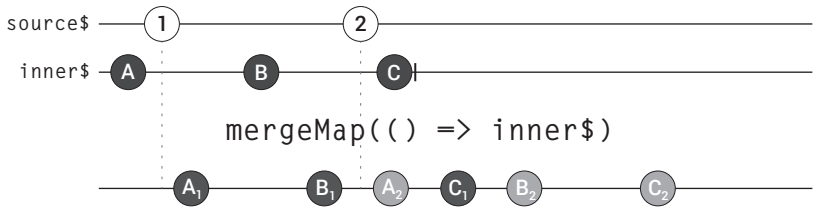


Abb. 10–10
Visualisierung von mergeMap()

Verschiedene Flattening-Operatoren

Der Operator `mergeMap()` leitet die Daten einfach genau so weiter, wie sie gerade eintreffen. Dieses Verhalten ist oft nicht das, was wir benötigen. Deshalb gibt es drei weitere Operatoren, die sich sehr ähnlich verhalten, aber subtile Unterschiede haben. Alle vier Kandidaten –

`mergeMap()`, `concatMap()`, `switchMap()` und `exhaustMap()` – haben gemeinsam, dass sie einen Datenstrom auf ein anderes Observable abbilden, Subscriptions auf die inneren Observables erstellen und die Ergebnisse zusammenführen. Die Unterschiede liegen in der Frage, ob und wann diese Subscriptions erzeugt werden. Jeder Operator setzt eine andere *Flattening-Strategie* um:

- `mergeMap`: Verwaltet mehrere Subscriptions parallel und führt die Ergebnisse in der Reihenfolge ihres Auftretens zusammen. Die Reihenfolge kann sich ändern.
- `concatMap`: Erzeugt die nächste Subscription erst, sobald das vorherige Observable completet ist. Die Reihenfolge bleibt erhalten, weil die Observables nacheinander abgearbeitet werden, wie in einer Warteschlange.
- `switchMap`: Beendet die laufende Subscription, sobald ein neuer Wert im Quelldatenstrom erscheint. Nur das zuletzt angefragte Observable ist interessant.
- `exhaustMap`: Ignoriert alle Werte aus dem Quelldatenstrom, solange noch eine Subscription läuft. Erst wenn die »Leitung« wieder frei ist, werden neue eingehende Werte bearbeitet.

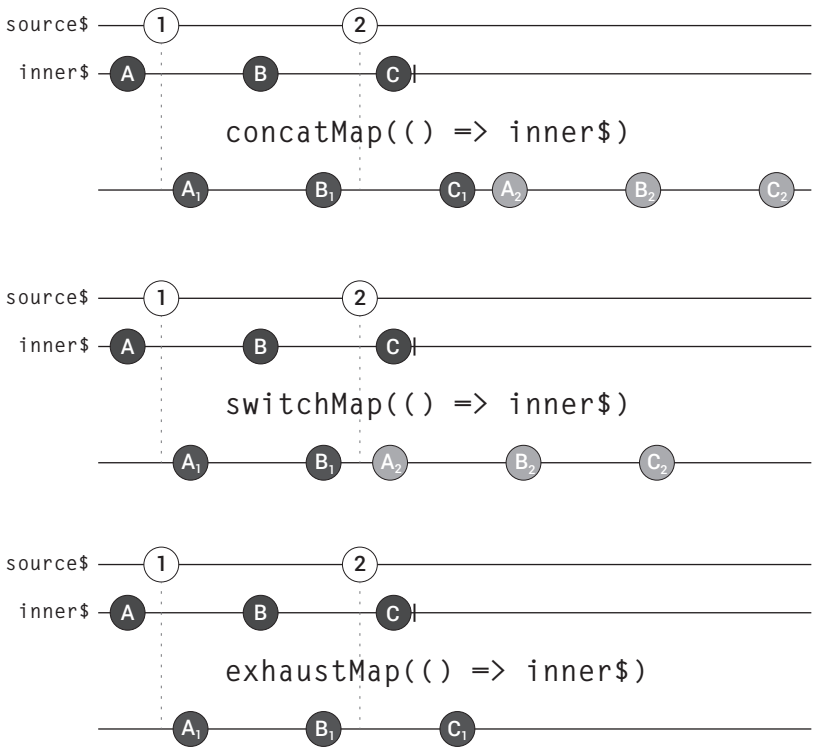


Abb. 10–11 *Visualisierung von* concatMap(), switchMap() *und* exhaustMap()

Teil V

Fortgeschrittene Themen

21 State Management mit Redux und NgRx

»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 nutzen dieselben Services. 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 oder 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 erkannt und verarbeitet werden müssen. Kurzum: Die Anwendung wird zunehmend schwerfälliger.

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 für die Anzeige zu lesen, neue Daten zu erfassen 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 das populäre Framework *Reactive Extensions for Angular (NgRx)*, um den Anwendungszustand zu verwalten und unsere Prozesse zu koordinieren.

21.1 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. Dieser *State* kann natürlich noch weitere Eigenschaften besitzen; wir haben dies mit dem Property `anotherProperty` angedeutet.

Listing 21–1
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 21–2
Zentralen Zustand in der Komponente verwenden

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

Den injizierten `StateService` können wir dann im Template nutzen[1], um die Daten anzuzeigen und die Methode `incrementCounter()` auszulösen:

[1] Services sollten nicht direkt im Template verwendet werden, um die Abhängigkeiten auf eine konkrete Implementierung zu verringern. Deshalb werden injizierte Services in der Regel `private` gesetzt. Um das vorliegende Beispiel einfach zu halten, verzichten wir hier allerdings darauf.

Listing 21–3 *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 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 das 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.

---

[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 unter »Wissenswertes« ab Seite 770 einen Abschnitt untergebracht.

[4] Im Kapitel zu reaktiver Programmierung mit RxJS haben wir Subjects ausführlich besprochen, siehe Seite 224.

Wir setzen im Service zunächst die Eigenschaft `state` auf privat, sodass man nun gezwungen ist, das Observable `state$` zu verwenden, anstatt direkt 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 21–4
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. So erhalten wir z. B. ein Observable, das nur den fortlaufenden Counter-Wert ausgibt.

Listing 21–5
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 21–6
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 Abonnenten 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 verschiedene Bibliotheken, darunter das Projekt *Immutable.js*[5] oder die leichtgewichtige Bibliothek *Immer*[6]. 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

[5] https://ng-buch.de/b/104 – Immutable.js

[6] https://ng-buch.de/b/105 – Immer

Für die meisten Anwendungsfälle 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[7] nutzen und damit alle Eigenschaften kopieren.

Im folgenden Listing 21–7 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 21–7
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`[8] einsetzen. Dies kann die Performance der Anwendung entscheidend verbessern.

Bitte beachten Sie, dass der Spread-Operator stets nur eine flache Kopie (Shallow Copy) eines Objekts erstellt. Ist ein Objekt oder Array verschachtelt, so müssen wir bei Änderungen auch immer das direkt betroffene Objekt kopieren.

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 aufge-

[7] Den Spread-Operator und die Rest-Syntax haben wir im Kapitel zu TypeScript ab Seite 42 erklärt.

[8] Auf die Change Detection und die Strategie `OnPush` gehen wir ab Seite 770 genauer ein.

rufen 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 signalisieren kann. Welche Routinen als Reaktion auf eine Nachricht anzustoßen sind, das entscheidet allein die Zustandsverwaltung. Die Komponenten teilen lediglich mit, was in der Anwendung passiert.

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. Die Methode `dispatch()` werden wir im nächsten Abschnitt noch genauer betrachten. Für den Moment vereinbaren wir, dass sie eine Nachricht entgegennimmt und für die gewünschte Zustandsänderung sorgt.

Listing 21–8 *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 zur (Pseudo-) Unveränderlichkeit im Hinterkopf wollen wir die Verwaltung des Zustands erneut überdenken. Bisher haben wir das State-Objekt direkt als Property 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.

Zustandslose Programmierung

Die Lösung des Problems ist, 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 diese ausgelagerten Funktionen keinen eigenen Zustand besitzen (engl. *stateless*), sodass sie bei gleichen Eingangswerten stets die gleichen Ausgangswerte erzeugen. Dadurch werden die Funktionen einfacher testbar.

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. Die Fallunterscheidung anhand der Nachricht können wir mit einer *switch/case*-Anweisung realisieren.

Listing 21–9
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 müssen in diesem Fall das vorherige State-Objekt unverändert zurückgeben. Unser zentraler Service kann also mithilfe der neuen Funktion wie folgt angepasst werden:

Listing 21–10
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, die Werte eines Arrays auf einen einzigen Wert zu reduzieren, indem für jeden Wert ein Callback ausgeführt wird. Die einfachste Form einer solchen Reduktion ist eine Summenbildung:

Listing 21–11 *Addition mit* Array.reduce()

```
const values = [1, 2, 3, 4];
const reducer = (previous, current) => previous + current;

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

Die Signatur unserer zuvor ausgelagerten Funktionen entspricht bereits einem solchen Callback, wie es auch für `Array.reduce()` verwendet wird. Unseren Zustand können wir demnach auch so berechnen: Es existiert ein Array von nacheinander abzuarbeitenden Nachrichten. Mithilfe von `Array.reduce()` summieren wir alle Nachrichten auf und verwenden dafür die Anweisungen aus der Funktion `calculateState()`.

Listing 21–12 *Nachrichten auf den Zustand reduzieren*

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

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

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 eine *Pure Function* ist. Sie liefert also für die gleichen Eingabewerte stets die gleiche Ausgabe und erzeugt keine Seiteneffekte. Dazu darf die Funktion ausschließlich die übergebenen Parameter verwenden und keinen eigenen Zustand verwalten. Wir gehen auf die Eigenschaften einer Pure Function später noch genauer ein.

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 neues 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 21–13 *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 bisherigen Zustand jederzeit wieder reproduzieren, sofern unsere Reducer-Funktionen deterministisch sind.[9] 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

[9] 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.

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.

21.2 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.

Store

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* 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

Neben dem zentralen Store mit dem State-Objekt verwendet Redux zwei weitere wesentliche Bausteine: 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 21–1 grafisch dargestellt. Hier ist gut er-

kennbar, dass die Daten stets in eine Richtung fließen und dass Lesen und Schreiben klar voneinander getrennt sind.

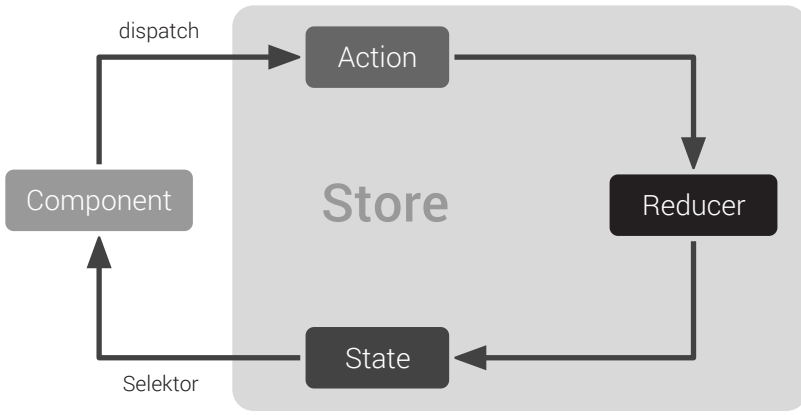


Abb. 21–1
Datenfluss in Redux

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 Frameworks und Bibliotheken, die ein zentrales State Management für Angular ermöglichen. Sie alle folgen der grundsätzlichen Idee von Redux.

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

NgRx ist das bekannteste Projekt aus dieser Kategorie. Das Framework wurde von Mitgliedern des Angular-Teams aktiv mitentwickelt und gilt als De-facto-Standard für zentrales State Management mit Angular. Es lohnt sich außerdem, einen Blick auf die Community-Projekte NGXS und Akita zu werfen. In der ersten Auflage dieses Buchs haben wir außerdem das Framework *angular-redux* vorgestellt. Leider wird das Pro-

jekt derzeit nicht weiterentwickelt, sodass wir seit der zweiten Ausgabe dieses Buchs auf NgRx setzen.

Welches der Frameworks 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]

21.3 NgRx: Reactive Extensions for Angular

Das Framework *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. NgRx 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 Abschnitt 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.

21.3.1 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/bm4-it7-i18n

[10] https://ng-buch.de/b/106 – Ordina JWorks Tech Blog: NGRX vs. NGXS vs. Akita vs. RxJS: Fight!

21.3.2 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 Angular-Projekt. Die folgenden Befehle integrieren einen vorbereiteten Store in die bestehende Anwendung:

```
$ 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 notwendige Paket in diesem Schritt gleich mit eingefügt. Die Store DevTools sind hilfreich zum Debugging der Anwendung – wir werden im Powertipp ab Seite 663 genauer darauf eingehen, um den Lesefluss in diesem Kapitel nicht zu unterbrechen.

21.3.3 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.

21.3.4 Grundstruktur

Die ausgeführten Befehle haben bereits alles Nötige eingerichtet, sodass wir sofort mit der Implementierung beginnen können. Vorher wollen

Teil VI

Angular-Anwendungen für Mobilgeräte

24 Progressive Web Apps (PWA)

»To me, it became clear that PWAs should be the future of software delivery.«

Sam Richard
(Developer Advocate für Chrome OS bei Google)

Mobilanwendungen sind mit der weiten Verbreitung von Smartphones und Tablets sehr populär geworden und ein wichtiges Instrument für die Außendarstellung eines jeden Unternehmens: Kunden wollen Apps, und fast jedes größere Unternehmen bietet für seine Produkte und Dienstleistungen inzwischen eine Mobilanwendung an. Der Begriff *App* fällt schnell im ersten Gespräch, wenn die Anforderungen an die Software definiert werden. Doch es muss tatsächlich nicht immer eine native App sein. Wenn wir keinen Zugriff auf tiefgreifende native Funktionen des Endgeräts benötigen, ist eine PWA eine sehr gute Alternative. Die Entwicklung einer PWA aus einer bestehenden Webanwendung heraus kann deutlich weniger Budget beanspruchen als die Neuentwicklung einer nativen App. Nach der Installation fühlt sich eine PWA für den Endnutzer an wie eine »echte App«.

Wir lernen in diesem Kapitel, wie wir unsere Angular-Anwendung in eine PWA verwandeln. Dazu schauen wir uns zunächst die wichtigsten Charakteristiken von PWAs an und widmen uns dem *Service Worker*. Anschließend integrieren wir diese Features in unseren BookMonkey.

24.1 Die Charakteristiken einer PWA

Wir wollen zunächst den Begriff der Progressive Web App etwas detaillierter einordnen. Bei einer PWA handelt es sich grundlegend auch um eine Webanwendung, sie wird allerdings durch den Nutzer heruntergeladen und auf dem lokalen Gerät gespeichert. Daraus ergibt sich, dass eine PWA nicht zwingend über einen App Store verteilt werden muss. Eine PWA kann Push-Benachrichtigungen erhalten und anzeigen, wie eine native Anwendung. Außerdem sorgt eine PWA dafür, dass Da-

ten im Client gecacht werden. Die Informationen bleiben mit der Anwendung also stets abrufbar, auch wenn ggf. keine durchgängige Internetverbindung vorhanden ist. Vielleicht kennen Sie das Konzept von einschlägigen Social-Media-Anwendungen: Beim Start der App sind zunächst die alten Beiträge aus dem Cache sichtbar, sogar wenn das Gerät offline ist. Können Daten vom Server nachgeladen werden, erscheinen wenig später die neuesten Inhalte.

Die drei wichtigsten Charakteristiken einer PWA sind also folgende:

- Hinzufügen zum Startbildschirm (»Add to Homescreen«)
- Offline-Fähigkeit
- Push-Benachrichtigungen

24.2 Service Worker

Service Worker sind kleine Helfer im Browser.

Als Grundvoraussetzung, um eine PWA offlinefähig zu machen und Push-Benachrichtigungen zu versenden, werden die sogenannten *Service Worker* benötigt. Diese werden von den meisten Browsern unterstützt, jedoch gibt es Ausnahmen wie den Internet Explorer.[1] Service Worker sind kleine Helfer des Browsers, die Aufgaben im Hintergrund übernehmen können. Ein Service Worker kann beispielsweise prüfen, ob eine Netzwerkverbindung besteht und passend dazu die Daten an die Anwendung ausliefern. Je nach Konfiguration werden die Daten aus dem Cache an die Anwendung übermittelt, oder der Service Worker versucht, die Daten online abzurufen. So können also Daten auf dem Endgerät zwischengespeichert werden. Eine weitere Aufgabe des Service Workers ist der Empfang von Push-Benachrichtigungen vom Server.

24.3 Eine bestehende Angular-Anwendung in eine PWA verwandeln

Wir wollen das Thema anhand eines Beispiels betrachten und den BookMonkey in eine PWA verwandeln. Die App kann auf dem Gerät installiert werden, und der Nutzer sieht stets Bücher in der Liste, auch ohne Internetverbindung.

Als Grundlage nehmen wir den finalen BookMonkey aus Iteration 7. Entweder verwenden Sie dafür Ihr eigenes Beispielprojekt, oder Sie klonen den vorbereiteten Stand von uns:

[1] https://ng-buch.de/b/119 – Can I Use: Service Workers

```
$ git clone https://ng-buch.de/bm4-it7-i18n.git book-monkey-pwa
$ cd book-monkey-pwa
$ npm install
```

Listing 24–1
Den BookMonkey als Grundlage für eine PWA nutzen

Als Nächstes fügen wir mithilfe von `ng add` das Paket `@angular/pwa` zum Projekt hinzu. Die dahinterliegenden Schematics nehmen uns bereits automatisch einen Großteil der Arbeit ab:

- Paket `@angular/service-worker` zu unserem Projekt hinzufügen
- Build Support für Service Worker in der Angular CLI aktivieren
- `ServiceWorkerModule` im `AppModule` importieren
- Datei `index.html` ergänzen: Link zum Web App Manifest (`manifest.json`) und relevante Meta-Tags
- Icon-Dateien erzeugen und verlinken
- Konfigurationsdatei `ngsw-config.json` für den Service Worker erzeugen

```
$ ng add @angular/pwa --project book-monkey
```

Listing 24–2
PWA einrichten

Unsere Anwendung ist nun bereit, um als PWA gestartet und genutzt zu werden. Wichtig ist, dass die Anwendung zum Testen der spezifischen PWA-Funktionalitäten immer im Produktivmodus gebaut werden muss, denn der Service Worker ist im Entwicklungsmodus nicht aktiv.

```
$ ng build --prod
```

Listing 24–3
Erstellen des Produktiv-Builds

Nach dem Build der Anwendung wollen wir uns das Ergebnis im Browser ansehen. Wir benötigen einen einfachen Webserver, der die Dateien ausliefert, z. B. aus dem Paket `angular-http-server`.

```
$ npx angular-http-server --path=dist/book-monkey
```

Listing 24–4
Den angular-http-server *zur Auslieferung des Projekts nutzen*

Die Besonderheit des `angular-http-server`

Der `angular-http-server` leitet im Gegensatz zum häufig eingesetzten `http-server` alle Anfragen zu nicht existierenden Verzeichnissen oder Dateien an die Datei `index.html` weiter. Das ist notwendig, da das Routing durch Angular und nicht durch den Webserver durchgeführt wird. Natürlich lassen sich auch andere Webserver so konfigurieren, dass sie auf dieselbe Art und Weise funktionieren. Wie Sie andere Webserver derart konfigurieren, erfahren Sie im Kapitel 18.7 ab Seite 555.

Die einfachste Strategie zum Testen der PWA ist, die Verbindung zum Server zu trennen, um das Caching-Verhalten des Service Workers zu beobachten. Dazu starten wir die Anwendung, sodass der Service Worker eingerichtet wird und die Daten cachen kann. Anschließend nutzen

wir die Google Chrome Developer Tools und aktivieren im Tab »Application« im Abschnitt »Service Workers« die Checkbox »Offline« (siehe Abbildung 24–1).

Nach dem Neuladen der Seite sollte die Anwendung weiterhin funktionieren: Die Startseite der App wird angezeigt, denn der Service Worker hat die Dateien gecacht. Navigieren wir allerdings zur Buchliste, so können keine Bücher angezeigt werden: Der HTTP-Request zur API schlägt fehl.

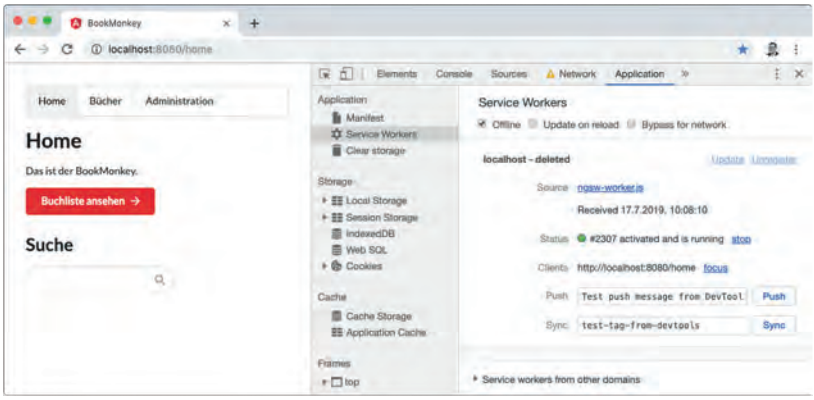


Abb. 24–1 *Offline-Modus in den Google Chrome Developer Tools aktivieren*

Service Worker funktionieren nur mit HTTPS oder localhost.

Nutzen wir einen Service Worker, muss die Anwendung immer über eine gesicherte Verbindung mit HTTPS oder über `localhost` aufgerufen werden. Wollen Sie also Ihre lokale IP-Adresse nutzen, um die App auf dem Handy zu öffnen, müssen Sie die Anwendung über HTTPS ausliefern.[2]

24.4 Add to Homescreen

Prinzipiell kann jede Website unter Android oder iOS zum Homescreen hinzugefügt werden. Sie erhält dann ein eigenes App-Icon und präsentiert sich dadurch wie eine native App. Unter iOS wird hierfür der Safari-Browser benötigt, unter Android funktioniert die PWA am besten unter Chrome.[3] Nach Bestätigung wird eine Verknüpfung auf dem

[2] https://ng-buch.de/b/120 – NPM: angular-http-server – Self-Signed HTTPS Use

[3] In Safari öffnen Sie mit dem Button »Teilen« (Rechteck mit Pfeil nach oben) das Menü und wählen darin die Option »Zum Home-Bildschirm«. In Chrome unter Android klicken Sie oben rechts auf die drei Punkte und wählen die Option »Zum Startbildschirm hinzufügen«. Die genaue Position und Bezeichnung dieser Optionen können sich je nach Version des Browsers unterscheiden.

Teil VII

Weiterführende Themen

28 Wissenswertes

»Being in the era of componentized web, one would love if a framework built for developing enterprise-level applications also supports the idea of having components rendered as standalone custom elements in the DOM. Angular allows creating a component as a Web Component using Angular Elements.«

Nishu Goel
(Google Developer Expert, Autorin und Softwareentwicklerin)

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.

28.1 Web Components mit Angular Elements

Trend: Komponentenbasierte Entwicklung

Das Rad nicht neu erfinden

Die Welt der Webentwicklung ändert sich stetig, und die Vielfalt von Anwendungen und deren Anforderungen nimmt täglich zu. Oft spezialisieren sich Teams auf ein Webframework und müssen dabei gleichzeitig bestehende Altanwendungen weiter betreiben. Seit Jahren ist hier ein gewisser Trend erkennbar: Die Entwicklung von Anwendungen basiert auf möglichst agnostischen, unabhängigen Komponenten. Dieser Trend wird vor allem dadurch beflügelt, dass Anwendungen möglichst schnell und featuregetrieben von mehreren Teams parallel entwickelt werden sollen. Man möchte auf bereits existierende Komponenten und Bibliotheken zurückgreifen, um das Rad nicht neu zu erfinden – sondern um sich auf die Implementierung der eigentlichen Geschäftslogik zu konzentrieren.

Die meisten Webanwendungen verwenden ein zugrunde liegendes Basisframework, das die Entwicklung der Anwendung erleichtert.

Frameworks verbessern die Developer Experience.

Vue.js[1], React.js[2], Svelte[3] und Angular sind nur einige dieser verbreiteten Frameworks. Alle haben eines gemeinsam: Sie erleichtern uns die Entwicklung von Anwendungen, indem sie bekannte Probleme lösen und uns einen wohldefinierten Rahmen mit Best Practices liefern. Außerdem stützen sich alle genannten Projekte auf das Konzept der komponentenbasierten Entwicklung. Die Komponenten können wir idealerweise entkoppelt von der konkreten Anwendung wiederverwenden. Dafür gibt es aber in der Regel eine Voraussetzung: Wir verwenden die Komponenten stets mit dem Framework, mit dem sie entwickelt wurden.

Komponentenbasierte Entwicklung

Arbeiten wir in nur einem Team oder einem Unternehmen, in dem die Frameworklandschaft homogen ist, stellt das in der Regel kein Problem dar, und wir können unsere einmal entwickelten Komponenten auch in anderen Anwendungen nutzen.

Frameworkabhängigkeit kann die Wiederverwendung behindern.

Ist die Landschaft im Unternehmen jedoch vielfältiger, so stellen sich neue Herausforderungen. Entwickelt ein Team beispielsweise eine spezielle Formularkomponente mit Angular, so kann ein anderes Team diese Komponente nur nutzen, wenn ebenfalls Angular zum Einsatz kommt. Arbeiten die Entwickler hier mit einem anderen Framework, ist die Formularkomponente nicht ohne weiteres nutzbar!

In der Praxis führt dies oft dazu, dass Parallelentwicklungen für ähnliche oder gleiche Features stattfinden. Ändern sich die Anforderungen oder gibt es Bugs, müssen diese in allen Implementierungen nachgezogen werden. Auch müssen viele gut durchdachte Features oder Konzepte für die Barrierefreiheit stets mehrfach durchdacht und in die verschiedenen Implementierungen integriert werden.

Die Antwort auf dieses technische Dilemma sind frameworkagnostische Komponenten, die völlig unabhängig vom Framework genutzt werden können. Die gemeinsame Plattform ist der Browser, daher spricht man hierbei von *Web Components*.

Web Components

Web Components sind agnostisch.

Web Components sind keine Neuheit in der Webentwicklung: Wir verstehen unter dem Begriff eine unabhängige Komponente, die agnostisch und unabhängig von einem konkreten Framework ist. Die Idee von Web Components wurde 2011 zum ersten Mal von Alex Russell auf einer Konferenz vorgestellt.[4]

---

[1] https://ng-buch.de/b/147 – Vuejs.org

[2] https://ng-buch.de/b/148 – Reactjs.org

[3] https://ng-buch.de/b/149 – Svelte.dev

[4] https://ng-buch.de/b/150 – Devopedia: Web Components

Die Grundbausteine für Web Components sind die folgenden:

Baustein	Beschreibung
HTML-Templates[5]	gruppieren Inhalte, die vom Browser zunächst nicht gerendert werden. Diese Inhalte können nur mittels JavaScript zur Laufzeit eingebunden werden.
Custom Elements[6]	sind selbstdefinierte HTML-Elemente, die die grundlegenden Elemente erweitern, die vom Browser interpretiert werden können.
Shadow DOM[7]	kapselt das Markup und den Style einer Web Component, sodass dieser von anderen Style-Definitionen und Komponenten isoliert wird.
ECMAScript-Module[8]	können mithilfe von JavaScript dynamisch Funktionen und Datenstrukturen importieren und exportieren.

Bis auf den Einsatz von Custom Elements haben wir in diesem Buch bereits drei der vier Bausteine grundlegend kennengelernt. Daher müssen wir nur noch eine Möglichkeit finden, eine Angular-Komponente als Web Component bereitzustellen. Mittlerweile unterstützen die meisten aktuellen Browser die Techniken von Web Components.[9] Müssen Sie einen älteren Browser unterstützen, z. B. den Internet Explorer 11, so können Sie hierfür einen Polyfill verwenden.[10]

Web Components mit Angular

Angular hilft bei der Erzeugung von Web Components.

Web Components sind unabhängig von einem Framework, und der Browser liefert die passende Plattform zur Unterstützung von Komponenten. Warum sollten wir nun überhaupt weiterhin ein Framework verwenden? Hierfür gibt es einige Antworten: Zum einen ist die Implementierung von Web Components mit reinem JavaScript (»VanillaJS«) im Vergleich zur Implementierung mit Angular oder einem anderen

[5] https://ng-buch.de/b/151 – Mozilla Developer Network: <template>

[6] https://ng-buch.de/b/152 – Mozilla Developer Network: Benutzerdefinierte Elemente

[7] https://ng-buch.de/b/153 – Mozilla Developer Network: Using shadow DOM

[8] https://ng-buch.de/b/154 – Mozilla Developer Network: JavaScript modules

[9] https://ng-buch.de/b/155 – Can I use: Web Components

[10] https://ng-buch.de/b/156 – Angular Docs: Browser support for custom elements

Framework recht aufwendig. Zum anderen sind wir mittlerweile Experten im Umgang mit Angular und TypeScript geworden – warum also auf der grünen Wiese beginnen und ggf. in Fehlersituationen laufen? Ein Framework liefert einen etablierten Rahmen zur Anwendungsentwicklung, der mehr ist als eine Reihe von Schnittstellen. Außerdem ist zu Projektstart oft noch gar nicht klar, ob Teile der Anwendung später überhaupt in anderen Projekten wiederverwendet werden müssen, die ggf. ein anderes Framework nutzen.

Die Schnittstelle zwischen dem Angular-Framework und den browsergetriebenen Web Components nennt sich *Angular Elements*. Dieses Modul liefert alles Nötige, um eine Angular-Komponente als Web Component bereitzustellen.

Angular Elements

Die Idee von Angular Elements ist leicht beschrieben: Eine bestehende Angular-Komponente wird als Grundlage verwendet, um eine Web Component zu erzeugen. Diese können wir anschließend mit herkömmlichem HTML und JavaScript nutzen oder sogar in ein anderes Webframework einbinden.

Eine Angular-Komponente in eine Web Component verwandeln

Um eine Komponente unserer Anwendung in eine Web Component zu verwandeln, benötigen wir im ersten Schritt die Toolunterstützung für Angular Elements in unserem Projekt.

Eine separate Anwendung für die Web Components

Die bestehende Angular-Anwendung beinhaltet bereits einen vollständigen Komponentenbaum, der mit der `AppComponent` beginnt. Diesen Baum möchten wir mit Angular Elements nun gerade nicht vollständig abbilden, sondern wir wollen nur einzelne dieser Komponenten herauslösen. Die Hauptanwendung soll weiterhin auch ohne Elements funktionieren, daher erzeugen wir innerhalb des Workspace eine neue Anwendung mit dem Namen `elements` – diesen Namen können Sie natürlich frei wählen.

```
$ ng g application elements --defaults
```

Die Anwendung wird in der Datei `angular.json` registriert und im Verzeichnis 🗀`projects/elements` angelegt.

Angular Elements zur Anwendung hinzufügen

Im nächsten Schritt fügen wir Angular Elements mithilfe der bereitgestellten Schematics in die neue Anwendung `elements` ein:

```
$ ng add @angular/elements --project=elements
```

Zunächst sollten wir die `AppComponent` der Anwendung `elements` komplett entfernen, denn dieses Projekt soll lediglich einzelne wiederverwendbare Komponenten beinhalten. Dazu entfernen wir den Eintrag

Index