Ferdinand Malcher · Danny Koppenhagen · Johannes Hoppe

Angular

Das große Praxisbuch –
Grundlagen, fortgeschrittene Themen
und Best Practices

inkl. RxJS, NgRx und a11y

iX EDITION

dpunkt.verlag

Leseprobe

Angular, 4. Auflage

https://angular-buch.com

Liebe Leserin, lieber Leser,

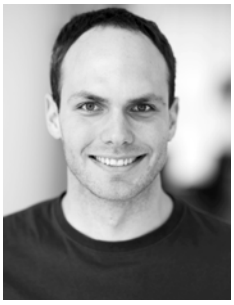
das Angular-Ökosystem wird kontinuierlich verbessert. Bitte haben Sie Verständnis dafür, dass sich seit dem Druck dieses Buchs unter Umständen Schnittstellen und Aspekte von Angular weiterentwickelt haben können. Die GitHub-Repositorys mit den Codebeispielen werden wir bei Bedarf entsprechend aktualisieren.

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 unter *team@angular-buch.com* mit einer Beschreibung des Problems.

Wir wünschen Ihnen viel Spaß mit Angular!

Alles Gute
Ferdinand, Danny und Johannes



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 Schulungen zu Angular an.

Danny Koppenhagen arbeitet als Softwarearchitekt und Entwickler. Sein Schwerpunkt liegt in der Frontend-Architektur und der Entwicklung von Enterprise Webanwendungen auf Basis von Node.js, TypeScript, Angular und Vue. Neben der beruflichen Tätigkeit ist Danny als Autor mehrerer Open-Source-Projekte aktiv.

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

Sie erreichen das Autorenteam auf Twitter unter *@angular_buch*.
Mehr Infos und Kontaktmöglichkeiten finden Sie unter *https://angular-buch.com/autoren*.

Ferdinand Malcher · Danny Koppenhagen · Johannes Hoppe

Angular

Das große Praxisbuch – Grundlagen, fortgeschrittene Themen und Best Practices

4., überarbeitete und aktualisierte Auflage

dpunkt.verlag

iX EDITION

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

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

Lektorat: René Schönfeldt
Projektkoordinierung: Anja Ehrlich
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: Schleunungdruck GmbH, Marktheidenfeld
Bindung: Hubert & Co. GmbH & Co. KG. BuchPartner, Göttingen

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 978-3-86490-946-7
PDF 978-3-96910-862-8
ePub 978-3-96910-863-5
mobi 978-3-96910-864-2

4., überarbeitete und aktualisierte Auflage 2023

Wieblinger Weg 17
69123 Heidelberg

Hinweis:
Der Umwelt zuliebe verzichten wir 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

Inhaltsübersicht

Inhaltsverzeichnis

IV Projektübergreifende Themen 507

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. Mit Angular in Version 2.0.0 setzte Google im Jahr 2016 einen Meilenstein in der Welt der modernen Webentwicklung: Das Framework nutzt die Programmiersprache TypeScript, bietet ein ausgereiftes Tooling und ermöglicht die komponentenbasierte Entwicklung von Single-Page-Anwendungen für den Browser und für Mobilgeräte.

React und Vue.js

In kurzer Zeit haben sich rund um Angular ein umfangreiches Ökosystem und eine vielfältige Community gebildet. Angular gilt neben React.js und Vue.js als eines der weltweit beliebtesten Webframeworks. Sie haben also die richtige Entscheidung getroffen, als Sie Angular für die Entwicklung Ihrer Projekte ins Auge gefasst haben.

Opinionated Framework

Der Einstieg in Angular ist umfangreich, aber die Konzepte sind durchdacht und konsequent. 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 gesamten Community einheitliche Konventionen herrschen, Standardlösungen existieren und bestehende Bibliotheken vorausgewählt wurden.

Beispielanwendung

Sie werden in diesem Buch lernen, wie Sie mit Angular komponentenbasierte Single-Page-Applikationen erstellen. Dazu entwickeln wir mit Ihnen gemeinsam eine Anwendung, anhand derer wir Ihnen die Konzepte und Features von Angular beibringen. Wir führen Sie Schritt für Schritt durch das Framework – vom Projektsetup über Komponenten, Routing, Formulare und HTTP bis hin zum Testing und Deployment der Anwendung. Auf dem Weg stellen wir Ihnen eine

Reihe von Tools, Tipps und Best Practices vor, die wir in mehr als sechs Jahren Praxisalltag mit Angular sammeln konnten. Die umfangreichen Theorieteile eignen sich auch später als Nachschlagewerk im Entwicklungsalltag.

Nach dem Lesen dieses Praxisbuchs 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 mit 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. Die Geschichte dieses Projekts reicht zurück bis ins Jahr 2009 zur Vorgängerversion *AngularJS*. Bis auf den ähnlichen Namen und einige Konzepte haben die beiden Frameworks aber nichts miteinander zu tun: Angular ab Version 2 ist eine vollständige Neuentwicklung und ist nicht mit dem alten AngularJS kompatibel.

It's just »Angular«.

Die offizielle Bezeichnung für das 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. Um Verwechslungen auszuschließen, gilt also die folgende Konvention:

- **Angular** – das Angular-Framework ab **Version 2 und höher** (dieses Buch ist durchgängig auf dem Stand von Angular 15)
- **AngularJS** – das Angular-Framework in der **Version 1.x.x**

AngularJS wird seit Januar 2022 offiziell nicht mehr weiterentwickelt und sollte nicht mehr verwendet werden.[1] Sie haben also die richtige Entscheidung getroffen, Angular ab Version 2.0.0 einzusetzen.

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

Die Versionsnummer *x.y.z* basiert auf *Semantic Versioning*.[2] Der Release-Zyklus von Angular ist kontinuierlich geplant: Im Rhythmus von ungefähr sechs Monaten erscheint eine neue Major-Version *x*. Die Minor-Versionen *y* werden monatlich herausgegeben, nachdem eine Major-Version erschienen ist. Jede Major-Version wird planmäßig für 1,5 Jahre unterstützt und weiterentwickelt (Long-Term Support).

Semantic Versioning

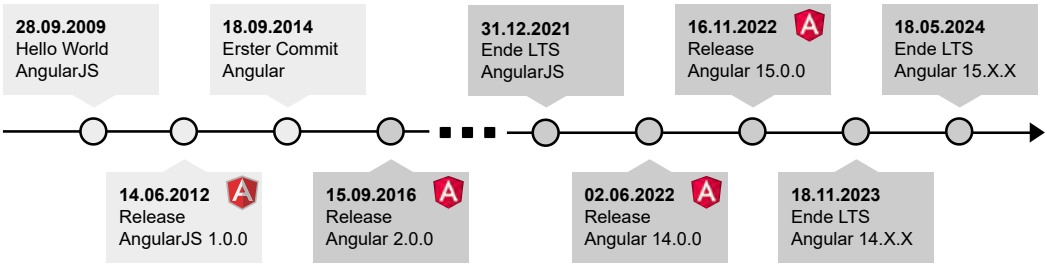


Abb. 1
Zeitleiste der Entwicklung von Angular

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 2.0.0 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.

Auf der Website zu diesem Buch finden Sie die Codebeispiele für das Beispielprojekt und viele weiterführende Informationen. Unter anderem veröffentlichen wir dort zu jeder Major-Version einen Artikel mit den wichtigsten Neuerungen in Angular. Wir empfehlen Ihnen aus diesem Grund, unbedingt einen Blick auf die Begleitwebsite zu werfen, bevor Sie beginnen, sich mit den Inhalten des Buchs zu beschäftigen:

Die Begleitwebsite zum Buch

https://angular-buch.com

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

An wen richtet sich das Buch?

Erfahrung in Softwareentwicklung

Dieses Buch richtet sich an Menschen, die bereits grundlegende Kenntnisse in der Softwareentwicklung mitbringen. Vorwissen zu JavaScript und HTML ist von Vorteil – es ist aber keine Voraussetzung, um mit diesem Buch Angular zu lernen. Wenn Sie jedoch bereits mit der Webentwicklung vertraut sind, werden Sie mit diesem Buch schnell starten können. Falls Sie gar keine Erfahrung in HTML und JavaScript mitbringen, empfehlen wir Ihnen, zunächst die grundlegenden Kenntnisse in diesen Bereichen zu festigen.

TypeScript

Für die Entwicklung mit Angular nutzen wir die populäre Programmiersprache TypeScript. Doch keine Angst: TypeScript ist eine Erweiterung von JavaScript, und die Konzepte sind sehr eingängig und schnell gelernt. Wenn Sie bereits eine stark typisierte Sprache wie Java oder C# kennen, wird Ihnen der Einstieg in TypeScript nicht schwerfallen.

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.

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. 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 mit den vielen praktischen Beispielen souverän meistern können.

Offizielle Angular-Dokumentation

Wir hoffen, dass dieses Buch Ihre tägliche Begleitung bei der Arbeit mit Angular wird. Für Details zu den einzelnen Framework-Funktionen empfehlen wir immer auch einen Blick in die offizielle Dokumentation.[3]

Wie ist dieses Buch zu lesen?

Einführung, Tools und Schnellstart

Im ersten Teil des Buchs lernen Sie die verwendeten Tools und die benötigten Werkzeuge kennen. Im Schnellstart tauchen wir sofort in Angular ein und nehmen Sie mit zu einem einfachen 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. Wenn Sie bereits Erfahrung im Umgang mit TypeScript

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

haben, können Sie diesen Teil auch überspringen und bei Bedarf später einzelne Themen nachlesen.

Beispielanwendung

Der dritte Teil ist der Hauptteil des Buchs: Hier werden wir mit Ihnen zusammen eine Beispielanwendung entwickeln. Die Konzepte und Technologien von Angular wollen wir dabei direkt am Beispiel vermitteln. Wir haben das Projekt in 17 einzelne Kapitel eingeteilt. In jedem Teil setzen wir gemeinsam mit Ihnen neue Anforderungen und technische Aspekte im Beispielprojekt um.

17 Praxiskapitel

- Komponenten: die Grundbausteine der Anwendung (ab S. 73)
- Property Bindings: mit Komponenten kommunizieren (ab S. 107)
- Event Bindings: Ereignisse in Komponenten verarbeiten (ab S. 123)
- NgModule: die Anwendung modularisieren (ab S. 141)
- Dependency Injection: Code in Services auslagern (ab S. 157)
- Routing: durch die Anwendung navigieren (ab S. 175)
- HTTP-Kommunikation: ein Server-Backend anbinden (ab S. 225)
- Reaktive Programmierung mit RxJS (ab S. 243)
- Interceptors: HTTP-Requests abfangen und transformieren (ab S. 301)
- Formulare mit Template-Driven Forms (ab S. 325)
- Formulare mit Reactive Forms (ab S. 345)
- Formularvalidierung: die Eingaben prüfen (ab S. 387)
- Pipes: Daten im Template formatieren (ab S. 413)
- Direktiven: das Vokabular von HTML erweitern (ab S. 433)
- Lazy Loading: Angular-Module asynchron laden (ab S. 459)
- Guards: Routen absichern (ab S. 471)
- Standalone Components: Komponenten ohne Module (ab S. 485)

Jedes dieser Kapitel besteht immer aus einem umfangreichen Theorieteil und der praktischen Implementierung im Beispielprojekt. Neben fachlichen Themen führen wir Refactorings durch, die die Architektur oder den Codestil der Anwendung verbessern. In mehreren *Powertipps* zwischen den Kapiteln zeigen wir außerdem hilfreiche Werkzeuge, die uns bei der Entwicklung zur Seite stehen.

Powertipps

Projektübergreifende Themen

Nachdem alle Praxiskapitel erfolgreich absolviert wurden, widmen wir uns einer Auswahl von projektübergreifenden Themen: Im Kapitel zu *Softwaretests* 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. Außerdem widmen wir uns ausführlich der *Barrierefreiheit*: In diesem Kapitel stellen wir Grundlagen und konkrete Maßnahmen vor, um die Anwendung für möglichst viele Menschen zugänglich zu machen. Zum Schluss werfen wir einen differen-

zierten Blick auf die *Lokalisierung* und *Internationalisierung*, um die Anwendung für den mehrsprachigen Betrieb vorzubereiten.

Deployment

Im fünften Teil des Buchs dreht sich alles um das Deployment einer Angular-Anwendung. Dabei betrachten wir die Hintergründe und Konfiguration des Build-Prozesses und erläutern die Bereitstellung mithilfe von Docker.

Fortgeschrittene Themen: NgRx, SSR und PWA

Im sechsten Teil möchten wir Ihnen einige Ansätze näherbringen, die über eine Standardanwendung hinausgehen. Hier stellen wir zunächst das *Redux*-Pattern und das populäre Framework *NgRx* vor, mit dem wir den Anwendungszustand zentral und gut wartbar verwalten können. Mit *Server-Side Rendering (SSR)* machen Sie Ihre Anwendung fit für Suchmaschinen und verbessern zusätzlich die wahrgenommene Geschwindigkeit beim initialen Start. Außerdem werfen wir einen ausführlichen Blick auf *Progressive Web Apps (PWA)*, um eine Webanwendung elegant auf Mobilgeräten zu nutzen.

Wissenswertes

Im letzten Teil »Wissenswertes« finden Sie weitere Informationen zu wissenswerten und begleitenden Themen. Hier haben wir weiterführende Inhalte zusammengetragen, auf die wir im Beispielprojekt nicht ausführlich eingehen.

Selbst tippen statt Copy & Paste

Der gesamte Code für das Beispielprojekt steht auf der Plattform GitHub zur Verfügung. Wir wissen genau, wie groß die Versuchung ist, größere Teile des Codes von dort zu kopieren und so die Tipparbeit zu sparen. Aber: Kopieren und Einfügen ist nicht dasselbe wie *Lernen* und *Verstehen*. Wenn Sie die Codebeispiele selbst *tippen*, 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!

Abtippen heißt Lernen und Verstehen.

Angular.Schule: Workshops und Beratung

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 effektivsten 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-Schulungen in den Räumen Ihres Unternehmens, in offenen Gruppen oder als Online-Kurs an. Das Angular-Buch verwenden wir dabei in unseren Kursen zur Nacharbeit. 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 die Geduld, unser Beispielprojekt zum vierten Mal durchzuarbeiten.

Wir bedanken uns bei **Mohammed Malekzadeh** und **Maximilian Franzke** für wertvolles Feedback und viele Anregungen zum Thema digitale Barrierefreiheit. **Jan Buchholz** danken wir für die hilfreichen Korrekturvorschläge. Wir danken **Lisa Möller** für die Zeichnungen zu unseren Personas (Seite 56). Ein großer Dank gilt außerdem unseren Familien, die uns auch abends an diesem Buch haben arbeiten lassen!

Dem Team vom dpunkt.verlag, insbesondere **Anja Ehrlich** und **René Schönfeldt**, danken wir für die Unterstützung und die 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 Entwicklungsalltag angenehmer macht. Zuletzt danken wir **Gregor Woiwode** für die Mitwirkung als Autor in der ersten Auflage. Ohne Gregor würde es das Angular-Buch in dieser Form nicht geben.

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

Aktualisierungen in der vierten 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 sechs 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.

Mit dieser Auflage haben wir die bislang größte und aufwendigste Überarbeitung gewagt: Wir haben uns mehr als ein Jahr Zeit genommen, um das Konzept dieses Buchs zu überdenken und auch Ideen und Baustellen zu bearbeiten, auf die wir bei den früheren Auflagen nicht den Fokus gesetzt hatten.

Alle Texte und Beispiele haben wir grundlegend überarbeitet und zum großen Teil neu verfasst. Dabei haben wir uns auch nicht davor gescheut, ganze Abschnitte zu löschen oder unsere Ideen aus der Vergangenheit kritisch zu hinterfragen.

Modernes und zeitloses Nachschlagewerk

Es ist unser Ziel, einen umfassenden Einstieg in das Angular-Framework zu ermöglichen – und gleichzeitig ein modernes und zeitloses Nachschlagewerk zu schaffen. Die Arbeit mit dem Buch dieser vierten Auflage lohnt sich daher auch für Leserinnen und Leser, die bereits eine der früheren Ausgaben besitzen. Diesen »frischen Wind der Veränderung« haben wir auch auf dem Buchcover mit einem Motiv aus der Raumfahrt aufgegriffen.

Angular 15 und folgende Versionen

Wir möchten Ihnen einen kurzen Überblick über die wichtigsten Neuerungen und Aktualisierungen der vierten Auflage geben. Alle Inhalte haben wir auf die Angular-Version 15 aktualisiert, sodass dieses Buch auch für die Arbeit mit den folgenden Versionen geeignet ist.

Neu in dieser Auflage

Wir haben das Buch neu strukturiert und das Beispielprojekt von Grund auf neu entwickelt. Dabei haben wir die Schwerpunkte anders gesetzt und die Erkenntnisse aus unserer täglichen Arbeit mit Angular

berücksichtigt. Die Praxisteile sind nun leichtgewichtiger und behandeln ausschließlich die Umsetzung im Beispielprojekt. Alle notwendigen Grundlagen werden jeweils in den umfassenden Theorieteilen behandelt. Besonders wichtige Aspekte haben wir als Merksätze hervorgehoben.

Um die Navigation im Buch zu vereinfachen, haben wir die frühere Gruppierung in »Iterationen« entfernt. Die Kapitel sind nun in einer **flacheren Struktur** organisiert.

Alle Inhalte und Beispiele sind auf dem **aktuellen Stand von Angular 15**. Dabei behandeln wir auch umfassend die neuesten Themen aus der Angular-Welt:

- **Standalone Components** sind ein neuer Ansatz, um Komponenten, Pipes und Direktiven unabhängig von Angular-Modulen in der Anwendung zu verwenden. Wir betrachten dieses Konzept ausführlich im neuen Kapitel 25 ab Seite 485.
- Die Bausteine für die Formularverarbeitung mit **Reactive Forms** sind seit Angular 14 stark typisiert. Kapitel 19 zu Reactive Forms ab Seite 345 behandelt diesen neueren Ansatz.
- Die neue **Funktion `inject()`** ist eine Alternative zur klassischen Constructor Injection, um Abhängigkeiten anzufordern. Sie kann in Zukunft ein elementarer Bestandteil der Arbeit mit Angular werden. Wir nutzen die Funktion an geeigneten Stellen, um die Verwendung zu vertiefen.
- **Interceptors, Guards und Resolvers** können als einfache Funktionen definiert werden. In früheren Versionen von Angular war dafür stets eine Klasse notwendig. Alle drei Bausteine werden nun ausführlich in der Theorie behandelt. Im Praxiskapitel zu den Guards ab Seite 480 implementieren wir auch einen funktionalen Guard.
- Mit der **Directive Composition API** können Direktiven »von innen« auf ein Element angewendet werden. Auf dieses neue Konzept gehen wir in Abschnitt 22.4 ab Seite 443 ein.
- Viele weitere neue Features und Aspekte des Angular-Frameworks haben wir über das Buch hinweg berücksichtigt.

In JavaScript wurde ein neuer Weg zur **Initialisierung von Klassen-Propertys** eingeführt, der sich vom proprietären Verhalten der Programmiersprache TypeScript unterscheidet. Unter »Wissenswertes« ab Seite 851 haben wir dieses Thema im Detail betrachtet. Der gesamte Code in diesem Buch ist so zukunftssicher ausgelegt, dass er in beiden Varianten problemlos funktioniert.

Für das Beispielprojekt stellen wir erstmals ein **eigenes Stylesheet-Paket** bereit. Zuvor hatten wir die Bibliothek *Semantic UI* genutzt, um die Anwendung zu gestalten. Mit einem eigenen Stylesheet für unser Projekt verkürzen sich nun die HTML-Templates, und der Fokus liegt mehr auf der tatsächlichen Arbeit mit Angular.

Das Thema der **digitalen Barrierefreiheit** ist ein wichtiger Bestandteil dieser Auflage. Neben einem neuen umfangreichen Kapitel zur Barrierefreiheit im Web ab Seite 577 haben wir auch das Beispielprojekt möglichst barrierearm umgesetzt.

Angular-Module sind ein wichtiger Baustein zur Strukturierung von Angular-Anwendungen. Anstatt das Thema erst zum Ende des Beispielprojekts zu behandeln, setzen wir nun schon frühzeitig in Kapitel 10 ab Seite 141 darauf, die Anwendung in Module zu teilen.

Die **AsyncPipe** ist ein elementarer Bestandteil von Angular, um die Arbeit mit Observables und Datenströmen zu vereinfachen. Obwohl wir Pipes erst in einem späteren Kapitel ausführlich behandeln, haben wir die AsyncPipe bereits in Kapitel 15 zu RxJS ab Seite 243 aufgegriffen. Im Beispielprojekt verwenden wir nun durchgehend die AsyncPipe, um Observables im Template aufzulösen. Damit orientieren wir uns an den etablierten Best Practices für Angular.

Die **Angular DevTools** sind ein umfangreiches Debugging-Werkzeug für Angular-Anwendungen. In einem neuen Powertipp ab Seite 321 gehen wir auf die Möglichkeiten dieser Browser-Extension ein.

Im Beispielprojekt setzen wir von vornherein auf das **Analysetool ESLint**, um einen einheitlichen Codestil zu ermöglichen und Best Practices technisch durchzusetzen, siehe Abschnitt 5.6 ab Seite 68.

Die Kapitel zur **Lokalisierung** (ab Seite 599) und **Internationalisierung** (Seite 605) haben wir thematisch getrennt und aus dem Beispielprojekt herausgelöst. Am praktischen Beispiel erläutern wir dort nun auch die Möglichkeit, Übersetzungen zur Laufzeit der Anwendung zu laden.

Das Test-Framework Protractor wird nicht mehr weiterentwickelt. Im Kapitel zu Softwaretests ab Seite 509 setzen wir deshalb für die Oberflächentests nun auf das etablierte Framework **Cypress**. Außerdem erläutern wir die Möglichkeit, einzelne Komponenten mithilfe von Cypress zu testen.

In Kapitel 32 zum Framework NgRx ab Seite 679 gehen wir auf den Architekturansatz der **Facades** ein. Außerdem betrachten wir das **Framework `@ngrx/component-store`**, mit dem wir den lokalen Zustand von Komponenten verwalten können.

Unter »Wissenswertes« ab Seite 827 geben wir einen kurzen Ausblick auf das populäre Build-Werkzeug **Nrwl Nx**, mit dem wir Monorepos effizient verwalten können. Außerdem werfen wir einen Blick auf das Tool **Storybook**, um die Komponenten der Anwendung in einem Katalog darzustellen.

Wir haben umfangreiche Rückmeldungen von unseren Leserinnen und Lesern erhalten. Jede eingegangene Anmerkung haben wir ausführlich diskutiert und bestmöglich im Buch umgesetzt. Ein Fachbuch wie dieses lebt von dem **Feedback aus der Community**, und wir freuen uns, dass so viele Menschen ihre Anregungen zum Buch mitgeteilt haben.

Wir haben uns in dieser Auflage zum ersten Mal für eine **geschlechtsneutrale Ansprache** entschieden. Wir freuen uns, dass wir so zumindest einen kleinen Teil dazu beitragen können, unsere alltägliche Sprache inklusiver zu gestalten.

Zu guter Letzt haben wir über vielen Kapiteln **Zitate** von Persönlichkeiten aus der Angular-Community aufgeführt. Die meisten dieser Stimmen haben wir direkt für dieses Buch erbeten. Wir freuen uns sehr, dass so viele persönliche, interessante und humorvolle Worte diesem Buch eine einmalige Note geben.

Fehler gefunden?

Neben den genannten Kapiteln haben wir alle Texte im Buch 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!

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 verwenden wir die Programmiersprache *TypeScript*.[1] Doch keine Angst – Sie müssen keine vollständig 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 leichter fällt, 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 TypeScript einsetzen

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.

[1] https://ng-buch.de/c/24 – TypeScript

Features aus allen 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. TypeScript vereint viele Features aus aktuellen und kommenden ECMAScript-Versionen, sodass wir sie problemlos auch für ältere Browser einsetzen können.

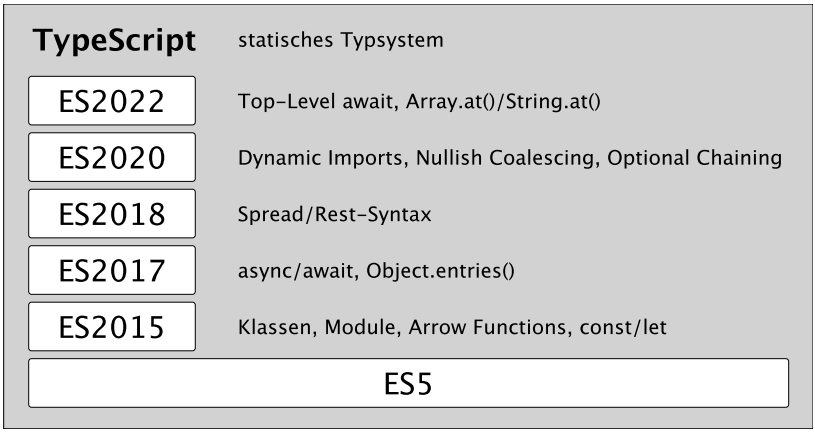


Abb. 4–1 *TypeScript und ECMAScript*

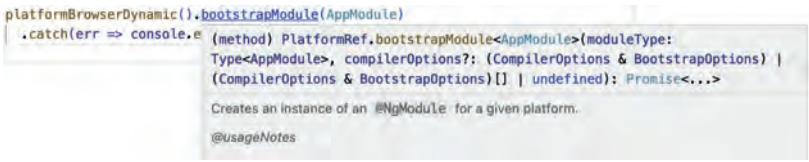


Abb. 4–2 *Unterstützung durch den Editor: Type Information On Hover*

TypeScript ist als Open-Source-Projekt bei der Firma Microsoft entstanden. 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.

Typisierung

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.

TypeScript und Angular

Wir möchten in den folgenden Abschnitten die wichtigsten Themen rund um ECMAScript und TypeScript vorstellen, damit Sie sicher mit der Sprache umgehen können.

4.2 Variablen: `const`, `let` und `var`

Ursprünglich wurden Variablen in JavaScript mit dem Schlüsselwort `var` eingeleitet. Das funktioniert noch immer, allerdings kamen mit ECMAScript 2015 die neuen Variablenarten `let` und `const` hinzu.

Die schmerzhafte `var`-heit

Mit dem Schlüsselwort `var` eingeleitete Variablen sind jeweils in der Funktion gültig, in der sie auch deklariert wurden – und zwar überall. Variablen mit `var` »fressen« sich durch alle Blöcke hindurch und sind in der *gesamten* Funktion und in allen darin verschachtelten Blöcken und Funktionen verfügbar. Das folgende Codebeispiel zeigt zwei Implementierungen, die zum exakt selben Ergebnis führen:

```
function foobar(foo) {
  if (foo) {
    var bar = 'angular';
  }
  // bar = 'angular'
};
```

Teil III

BookMonkey 5:
Schritt für Schritt zur App

6 Komponenten: die Grundbausteine der Anwendung

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

Igor Minar
(ehemaliges Mitglied des Angular-Teams)

Unsere Projektumgebung ist vorbereitet, also können wir nun beginnen, die ersten Schritte im BookMonkey zu implementieren. Wir betrachten in diesem Kapitel das Grundkonzept der Komponenten in Angular. 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 Komponenten

Komponenten sind die Grundbausteine einer Angular-Anwendung, und jede Anwendung ist aus vielen verschiedenen Komponenten zusammengesetzt. Eine Komponente beschreibt immer einen Teil der Oberfläche, z. B. eine Seite, einen Teilbereich der Seite oder ein einzelnes UI-Element. In der Regel wird jeder funktional abgrenzbare Teil der Oberfläche durch eine Komponente beschrieben.

Komponente: funktional abgrenzbarer Teil der UI

Eine Komponente besitzt dafür immer ein Template: Es ist das »Gesicht« der Komponente, also der Bereich, der im Browser sichtbar dargestellt wird. Es wird in der Regel in HTML notiert. Dazu kommt eine TypeScript-Klasse, in der wir die Logik für die Komponente definieren. Eine Komponente besteht immer aus diesen beiden Teilen: Sie bilden einen festen Verbund und können miteinander kommunizieren, um Daten und Events auszutauschen.

Jede Anwendung besitzt mindestens eine Komponente, die Hauptkomponente (engl. *Root Component*) mit dem Namen `AppComponent`. Alle weiteren Komponenten sind dieser Hauptkomponente untergeordnet.

Hauptkomponente

Komponente: Klasse mit Decorator @Component()

Technisch ist eine Komponente immer eine TypeScript-Klasse, die mit dem Decorator `@Component()` versehen ist. Darüber werden verschiedene Metadaten an die Klasse angehängt, unter anderem können wir in der Eigenschaft `template` das Template für die Komponente definieren. Das Listing 6–1 zeigt den Grundaufbau einer Komponente.

Listing 6–1 *Eine simple Komponente*

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

Selektor

Eine besondere Rolle spielt der Selektor, der im Property `selector` angegeben wird: Damit die Komponente sichtbar wird, muss sie in einem DOM-Element gerendert werden. Der Selektor legt fest, in welchen DOM-Elementen eine Instanz der Komponente erzeugt wird. Theoretisch können wir hier einen beliebigen CSS-Selektor angeben, für Komponenten werden jedoch immer nur Element-Namen verwendet.

Host-Element

Verwenden wir also das Element `<my-component></my-component>` in einem anderen Template unserer Anwendung, so erzeugt Angular in diesem Element eine Instanz der oben gezeigten `MyComponent`. Ein solches Element wird als *Host-Element* bezeichnet. Es beherbergt eine Instanz der Komponente – mitsamt ihrer Logik und ihrem Template. Das Template der Komponente wird in das Host-Element eingesetzt und ist dann dort sichtbar.

Ein Präfix verwenden

Die Selektoren von Komponenten und Direktiven sollten immer mit einem Präfix versehen werden. Damit vermeiden wir Konflikte mit anderen Elementen, die eventuell dieselbe Bezeichnung verdient hätten.

Beim Anlegen des Projekts mit dem Befehl `ng new` haben wir die Option `--prefix=bm` gesetzt und so das Präfix auf den Wert `bm` festgelegt. Standardmäßig wird der Wert `app` verwendet. Diese Einstellung finden wir auch in der Datei `angular.json` wieder. Wenn wir Komponenten oder Direktiven mit der Angular CLI anlegen, wird das Präfix automatisch im Selektor berücksichtigt, z. B. `bm-root`.

Bei der Arbeit mit Angular erzeugen wir also regelmäßig verschiedene Komponenten, die jeweils einen kleinen Teil der Logik und der Oberfläche beschreiben. Mithilfe des Selektors binden wir die Komponenten dann in die Templates von anderen Komponenten ein. Auf diese Weise können wir Komponenten beliebig tief verschachteln und komplexe Oberflächen beschreiben. Diese Praxis schauen wir uns im nächsten Kapitel ab Seite 107 genauer an. Komponenten sind wiederverwendbar, sodass ein einmal definierter Baustein mehrfach für die gleiche Aufgabe eingesetzt werden kann.

Das Template einer Komponente

Eine Komponentenklasse ist immer mit einem Template verknüpft. Das Template ist der sichtbare Teil der Komponente, mit dem wir in der Oberfläche interagieren können. Für die Beschreibung wird in der Regel HTML verwendet[1], denn wir wollen unsere Anwendung ja im Browser ausführen. In den Templates wird 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:

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

Welchen dieser beiden Wege wir wählen, hängt vom Geschmack und von der Größe des Templates ab. Für kleine Templates kann ein Inline-Template lohnenswert sein – so hat man alle Dinge sofort auf einen Blick verfügbar. Wird das Template zu lang, empfehlen wir hingegen, es in eine eigene Datei auszulagern. Die Angular CLI legt das Template standardmäßig auch in einer separaten Datei ab. Dafür verwenden wir die Eigenschaft `templateUrl` im Decorator `@Component()`. In Listing 6–2 sind beide Varianten zur Veranschaulichung aufgeführt. Die gezeigte Kombination funktioniert natürlich nicht, denn eine Komponente hat immer genau ein Template.

Listing 6–2
Template einer Komponente definieren

```
@Component({
  // Referenz zu einem HTML-Template
  templateUrl: './my-component.html',
  // ODER: HTML-String direkt im TypeScript
  template: `<h1>Hello 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 an-

[1] Statt HTML können wir auch SVG verwenden, um eine Komponente zu bauen, die eine Vektorgrafik darstellt. Dieser Fall ist aber relativ selten – normalerweise wird das Template in HTML notiert.

gezeigt werden. Umgekehrt können Ereignisse im Template abgefangen werden, um von der Komponente verarbeitet zu werden. Diese Kommunikation ist schematisch in Abbildung 6–1 dargestellt.

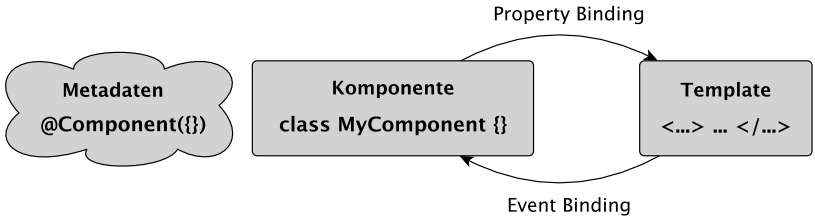


Abb. 6–1
Komponente, Template und Bindings im Zusammenspiel

Um diese Bindings zu steuern, nutzen wir die Template-Syntax von Angular, die wir gleich noch genauer betrachten. In den beiden folgenden Kapiteln gehen wir gezielter auf die beiden Arten der Bindings ein:

- Property Bindings: mit Komponenten kommunizieren (ab Seite 107)
- Event Bindings: Ereignisse in Komponenten verarbeiten (ab Seite 123)

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: SCSS/Sass und Less werden direkt unterstützt. Welches Style-Format wir standardmäßig verwenden wollen, können wir direkt auswählen, wenn wir die Anwendung mit `ng new` erstellen. Die Einstellung können wir später in der Datei `angular.json` ändern.

Style-Definitionen können wir auf zwei Wegen in die Anwendung bringen:

- als globales Stylesheet, das Styles für Elemente der gesamten Anwendung definiert
- als Style einer Komponente, in dem lediglich Aspekte der Komponente selbst gestylt werden können

Gloable Stylesheets vs. Komponenten-Styles

Definieren wir spezifische Styles für Komponenten auf globaler Ebene, kann es leicht unübersichtlich werden, und das Prinzip der Modularität ginge verloren. Auf der anderen Seite gibt es jedoch Style-Definitionen, die für die gesamte Anwendung gelten sollen, z. B. die verwendeten Schriftarten, Farben, Abstände oder die Standard-Styles von Buttons und Input-Feldern. Solche Styles ergeben auf Komponentenebene oft

wenig Sinn, und wir empfehlen, diese im globalen Stylesheet unterzubringen.

Stylesheets von Komponenten sind isoliert.

Eine direkte Verknüpfung von Styles und Komponenten sorgt außerdem 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. Die Technik hinter dieser Isolation nennt sich *View Encapsulation*.[2]

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

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

In Listing 6–3 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 zu verwenden, die in der Komponente referenziert wird.

Listing 6–3
Style-Definitionen in Komponenten

```
@Component({
  // Style-Definitionen direkt im TypeScript
  styles: [
    'h2 { color: blue; }',
    'h1 { font-size: 3em; }'
  ]
  // ODER: Referenz zu Style-Dateien
  styleUrls: ['./my.component.css'],
})
export class MyComponent { }
```

Der herkömmliche Weg zum Styling der Anwendung ist natürlich trotzdem weiter möglich: Wir können globale CSS-Dateien definieren, die in der gesamten Anwendung gelten und nicht nur auf Ebene der Kom-

[2] Ein Blick hinter die Kulissen: Jedes DOM-Element erhält automatisch ein zusätzliches Attribut mit einem eindeutigen Bezeichner. Die Stylesheets der Komponente werden beim Build ebenso erweitert, sodass sie auf dieses Attribut matchen. So sorgt Angular mit den Mitteln von CSS dafür, dass die Styles nur auf das Template der Komponente wirken. Neben diesem Standardverhalten gibt es weitere Strategien für die View Encapsulation, die bei Bedarf aktiviert werden können.

ponenten. Diesen Weg haben wir gewählt, um das Stylesheet aus dem Paket `book-monkey5-styles` einzubinden, siehe Seite 66. Es enthält komponentenübergreifende Style-Definitionen, die für die gesamte Anwendung relevant sind. Die meisten CSS-Frameworks und externen Komponentenbibliotheken funktionieren auf diese Weise.

6.2 Komponenten in der Anwendung verwenden

Wir wollen das Gelernte kurz zusammenfassen: Eine Komponente wird immer in einer eigenen TypeScript-Datei notiert. Dazu kommen meist ein separates Template, eine Stylesheet-Datei und eine Testspezifikation (`.spec.ts`). Diese vier Dateien sollten immer gemeinsam in einem eigenen Ordner untergebracht werden. So wissen wir sofort, welche Dateien zu der Komponente gehören. Ein Sonderfall ist die Hauptkomponente `AppComponent`: Sie liegt ohne einen Unterordner direkt in 🗀`src/app`.

Eine Komponente besitzt einen Selektor. In jedem DOM-Element, das zu diesem Selektor passt, wird automatisch eine Instanz der Komponente erzeugt. Das Element wird das Host-Element der Komponente.

Komponenten deklarieren

Damit dieser Mechanismus funktioniert, muss Angular die Komponente allerdings erst kennenlernen. Dazu muss eine Komponente in der Regel in einem NgModule deklariert werden.[3] Im Decorator `@NgModule()` können wir eine Liste von `declarations` notieren: Hier werden alle Komponenten[4] angegeben, die zu diesem Modul gehören. Damit wir die Typen dort verwenden können, müssen wir sie im Kopf der Datei importieren.

Zu Beginn besitzt eine Anwendung nur ein einziges Modul, das `AppModule`. Wir werden uns aber später in Kapitel 10 ab Seite 141 noch ausführlicher mit Modulen auseinandersetzen und weitere NgModules anlegen.

Listing 6–4 *Komponenten im* AppModule *deklarieren*

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

import { AppComponent } from './app.component';
import { MyComponent } from './my/my.component';
import { FooComponent } from './foo/foo.component';
```

[3] Eine Ausnahme sind die Standalone Components, die auch ohne Deklaration in einem Modul funktionieren. Dieses Thema behandeln wir in Kapitel 25 ab Seite 485.

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

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

6.3 Komponenten generieren mit der Angular CLI

Da wir regelmäßig mit Komponenten arbeiten werden, wäre es sehr aufwendig, alle Dateien, Ordner und Verweise manuell anzulegen. Die Angular CLI, mit der wir bereits das Projekt generiert haben, bietet deshalb eine Reihe von Generatoren an, mit denen wir die Bausteine der Anwendung automatisch erzeugen können.

Um eine Komponente zu generieren, verwenden wir den folgenden Befehl:

```
$ ng generate component foo
```

Listing 6–5
Komponente generieren mit der Angular CLI

Die Komponente wird in einem eigenen Unterordner mit den vier dazugehörigen Dateien generiert: TypeScript-Klasse, HTML-Template, Stylesheet und eine Testspezifikation mit der Endung `.spec.ts`. Das Suffix `Component` wird automatisch an den Klassennamen angehängt. In diesem Beispiel wird also eine `FooComponent` erzeugt. Die neue Komponente wird außerdem im passenden Modul deklariert, in diesem Fall im `AppModule`.

Wir können uns ein wenig Tipparbeit sparen, wenn wir statt der ausgeschriebenen Parameter deren Aliase verwenden. Alle folgenden Varianten führen zum gleichen Ergebnis:

```
$ ng generate component foo
$ ng g component foo
$ ng generate c foo
$ ng g c foo
```

Listing 6–6
Komponente generieren mit Kurzbefehl

Es gibt übrigens keinen Befehl, um eine generierte Komponente wieder zu löschen. Die Ausführung eines Befehls sollte also gut überlegt sein. Im Zweifel können wir mithilfe einer Versionsverwaltung wie Git die Änderungen aber wieder zurücksetzen.

Außerdem bietet die Angular CLI für alle `generate`-Befehle einen Trockendurchlauf an. Dabei wird das Dateisystem nicht verändert, sondern es wird nur auf der Kommandozeile ausgegeben, was passieren wird.

Listing 6–7 *Komponente generieren (Dry Run)*

```
$ ng generate component foo --dry-run
```

Dieses Feature ist vor allem bei der Arbeit in großen Projekten wertvoll. Bevor wir etwas generieren, können wir uns so vergewissern, dass wir alle Parameter korrekt gesetzt haben. Wenn Sie sich unsicher sind, was ein Befehl genau tun wird, nutzen Sie bitte den Dry-Run.

6.4 Umgang mit Propertys von Komponenten

Wenn wir Daten in einer Komponente anzeigen wollen, müssen wir diese vorher in der TypeScript-Klasse in einem Property ablegen, nur dann können sie vom Template aus erreicht werden. Wie genau wir Daten im Template anzeigen, lernen wir im nächsten Abschnitt zur Template-Syntax.

Jedes Property sollte sofort einen Startwert besitzen. Wir können den Wert direkt bei der Deklaration zuweisen oder die Initialisierung im Konstruktor der Klasse durchführen:

```
export class MyComponent {
  foo = 'Angular';
  bar: number;
  title: string; // FEHLER

  constructor() {
    this.bar = 42;
  }
}
```

Tun wir das nicht, erhalten wir die folgende Fehlermeldung von TypeScript:

```
Property 'title' has no initializer and is not definitely
assigned in the constructor.
```

Bei primitiven Typen wie `string` oder `number` können wir meist problemlos einen Startwert festlegen. Er kann später durch die Logik der Klasse wieder überschrieben werden. Auch ein Property mit einem Array kann einfach mit einem leeren Array initialisiert werden, falls die Daten beim Start noch nicht vorliegen.

Schwieriger ist es, einen Startwert für ein komplexes Objekt anzugeben: Häufig können wir ein Property nicht initialisieren, oder es ist Teil der Anwendungslogik, dass es explizit auch *keinen Wert* besitzt. In diesem Fall müssen wir das Property als optional markieren, indem wir ein Fragezeichen (`?`) hinter den Namen setzen. Neben dem definierten Typ `string` kann das Property `title` nun auch jederzeit den Wert `undefined` annehmen.

```
export class MyComponent {
  // entspricht: string | undefined
  title?: string;
}
```

Bei jeder Verwendung eines optionalen Propertys müssen wir im Code prüfen, ob der Wert tatsächlich existiert oder `undefined` ist. Dafür können wir im TypeScript-Code z. B. eine `if`-Anweisung verwenden. Wie wir im Template einer Komponente mit optionalen Propertys umgehen, lernen wir im Verlauf der nächsten Abschnitte.

Strict Mode

Verantwortlich für dieses Verhalten ist übrigens der *Strict Mode* von TypeScript, der in neuen Projekten seit Angular 12 standardmäßig aktiviert ist. Früher konnten wir ein Property auch ohne strikte Initialisierung verwenden. Das widerspiegelte allerdings nicht die tatsächlichen Verhältnisse und konnte zu Laufzeitfehlern führen. Verwenden Sie Klassen-Propertys also immer so, wie wir es in diesem Abschnitt erklärt haben.

Merke: Ein Property muss immer entweder explizit initialisiert werden (direkt oder im Konstruktor), oder es muss als optional markiert werden.

Non-Null Assertion

Eine weitere Möglichkeit ist, eine sogenannte *Non-Null Assertion* zu verwenden, indem wir ein Ausrufezeichen (`!`) hinter das Property setzen. Damit lockern wir die Typisierung: Der Compiler nimmt an, dass immer der festgelegte Typ vorliegt und dass die Typen `null` und `undefined` in diesem Property nicht auftreten werden. Tatsächlich besitzt ein Property ohne Startwert aber zunächst den Wert `undefined`. TypeScript wird diesen Umstand ignorieren, und wir haben potenziell fehleranfälligen Code geschrieben. Verwenden Sie deshalb die Non-Null Assertion nur, wenn Sie genau wissen, was Sie tun. Wir empfehlen Ihnen, dieses Konstrukt niemals für Propertys von Klassen zu verwenden.

Nicht für Klassen-Propertys verwenden!

15 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 for 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.

15.1 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.[1] Sie haben einen Strom von wiederkehrenden Ereignissen

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

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 in die richtige Richtung.

Telefonieren und Chatten

Im Bus klingelt das Handy, Sie gehen ran und sprechen mit dem Anrufenden. Beide Personen 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 unwichtiger. 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 leckeren belegten Brot. Was ist passiert? Ein Strom von eingehenden Menschen, die Geld besitzen, wurde umgewandelt in einen Strom von ausgehenden Menschen, die nun Backwaren haben. Die Angestellten in der Bäckerei haben den Personenstrom abonniert und die einzelnen Menschen mit Lebensmitteln 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 Wet-

ter. Unsere Aufgabe ist es, diese Ereignisse zu beobachten und darauf zu reagieren. Sind wir nicht an den Ereignissen interessiert, finden sie trotzdem statt.

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: Personen interagieren mit der Anwendung, klicken auf Buttons und füllen 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 wiederkehrende 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.

15.2 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 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 Body 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 15–1 *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 15–2 *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. Der senkrechte Strich signalisiert hier den Aufruf von `complete()`.

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

Listing 15–3
Ausgabe des Programms

Abb. 15–1
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 drei Callback-Funktionen entgegennimmt, die in einem Objekt gebündelt sind. 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 die Routine. Daraus folgt, dass *nichts* passiert, wenn niemand die Funktion aufruft. Starten wir die Funktion hingegen mehrfach, so wird die Routine 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.

15.3 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 verschiedene Programmiersprachen entstanden. Der wichtigste Datentyp von Rx, das *Observable*, ist sogar mittlerweile ein Vorschlag für ECMAScript[2] geworden. RxJS ist der Name der JavaScript-Implementierung von ReactiveX.

[2] https://ng-buch.de/c/39 – GitHub: TC39 Observables for ECMAScript

ables zusammen. Das Resultat ist ein `Observable<T>`, das für jedes Event aus dem Trigger `source$` die Werte aus dem gemappten Observable `inner$` enthält. In Abbildung 15–9 haben wir den gesamten Sachverhalt zum besseren Verständnis visualisiert.

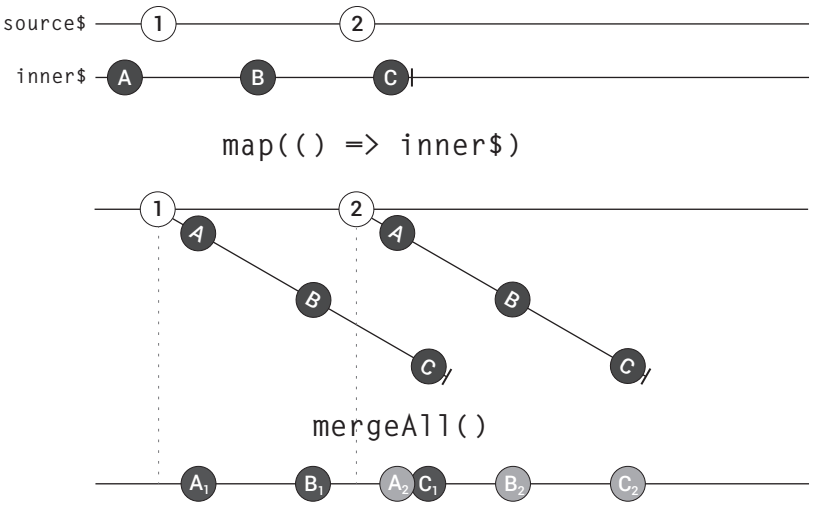


Abb. 15–9 *Flattening mit* mergeAll()

Diese Kombination von `map()` und `mergeAll()` ist so allgegenwärtig, dass es dafür einen eigenen Operator gibt.

Listing 15–38 mergeMap() *verwenden*

```
import { mergeMap } from 'rxjs';

source$.pipe(
  mergeMap(() => inner$)
);
```

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.

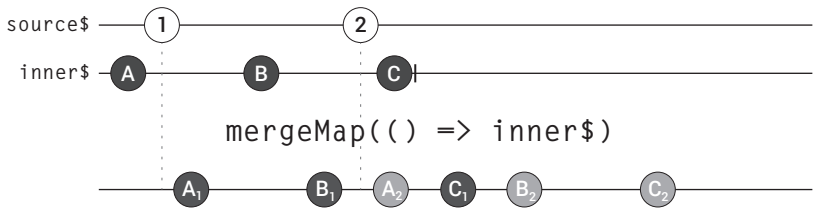


Abb. 15–10 *Visualisierung für* mergeMap()

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.

Um diese vier Operatoren korrekt einsetzen zu können, braucht es ein wenig Übung und Erfahrung. Im Zweifel sind Sie aber gut beraten, wenn Sie zunächst `concatMap()` verwenden, es sei denn, Sie haben einen guten Grund für einen der anderen Operatoren.

Beispiel: Sushi-Restaurant

Damit Sie diese Thematik besser verinnerlichen können, wollen wir wieder ein Beispiel aus dem echten Leben heranziehen. Sicher kennen Sie das Running Deck im Sushi-Restaurant, wo Teller mit kleinen Portionen vorbeifahren. Dieses Laufband ist ein (heißes) Observable (`sushiBelt$`), das einen Strom von Tellern liefert, die unterschiedliche Inhalte haben.

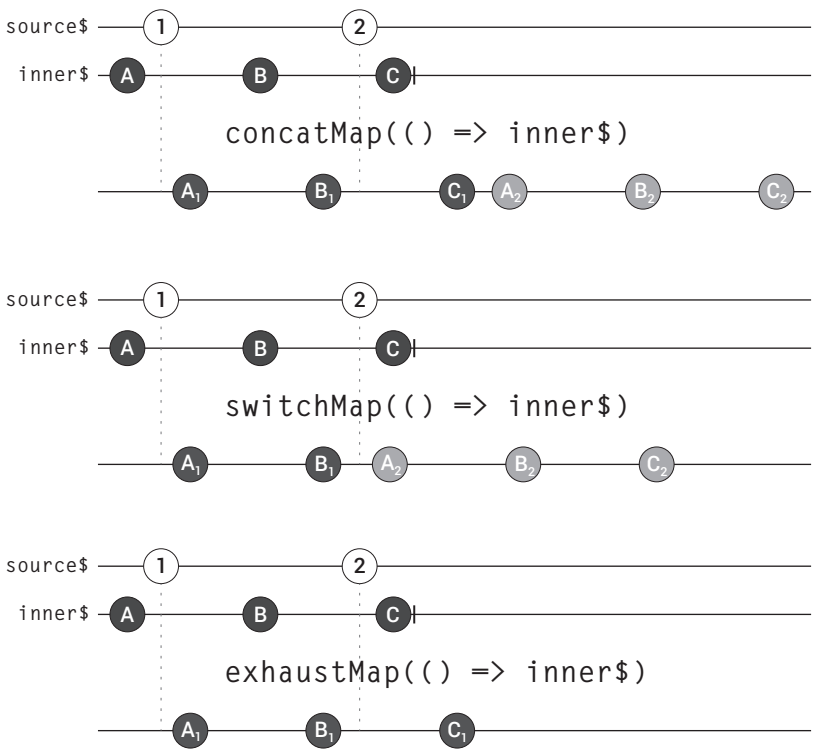


Abb. 15–11 *Visualisierung für* concatMap(), switchMap() *und* exhaustMap()

Auswahl treffen

Unsere Aufgabe ist es zunächst, aus diesem Strom von Tellern eine gute Auswahl zu treffen. Der Operator `filter()` entscheidet also mithilfe der Funktion `iWantThis(plate)`, ob wir einen Teller haben wollen oder nicht. Wir erzeugen damit einen Strom von Tellern, die wir tatsächlich vom Band nehmen.

Teller verarbeiten

Sobald die Teller auf dem Tisch stehen, müssen die Sushi-Röllchen verarbeitet werden. Da die Nahrungsaufnahme nicht synchron funktioniert, sondern Zeit benötigt, wollen wir diese Aktion wieder in einem Observable abbilden. Die Funktion `getSushiFromPlate(plate)` nimmt einen Teller entgegen und erzeugt ein (kaltes) Observable, das die enthaltenen Sushi-Röllchen nacheinander ausgibt.

Diese beiden Bausteine müssen wir nun zusammenfügen und verwenden dafür einen Flattening-Operator. Jeder vom Band genommene Teller wird also umgewandelt in einen kurzen Strom von einzelnen Sushi-Röllchen. Alle diese Teilströme werden in einem einzigen großen Sushi-Strom zusammengeführt – lecker!

25 Standalone Components: Komponenten ohne Module

»Dem mentalen Modell zufolge sind Standalone Components eigentlich nur Komponenten mit ihrem eigenen NgModule. Es ist so zwar nicht implementiert, aber man kann es sich so vorstellen.«

Manfred Steyer
(Google Developer Expert, Berater und Trainer für Angular)

Wir haben in den letzten 16 Kapiteln viele Aspekte von Angular kennengelernt und praktisch im BookMonkey umgesetzt. Dabei haben wir unsere Anwendung stets mit Modulen aufgebaut: Komponenten, Pipes und Direktiven werden in einem NgModule deklariert und können dann dort verwendet werden. Für das letzte Praxiskapitel mit dem BookMonkey haben wir uns ein besonderes Thema aufgehoben: Seit Angular 14 können wir mithilfe der *Standalone Components* unsere Komponenten, Pipes und Direktiven auch ohne ein Modul nutzen. Wir wollen die Ideen von Standalone Components genauer unter die Lupe nehmen und auch einige Teile des BookMonkey auf den neuen Ansatz migrieren.

25.1 NgModule und Standalone Components

Angular-Module mit NgModule sind ein fester Bestandteil des Frameworks, um Anwendungen zu strukturieren. Mithilfe von Modulen können wir vor allem fachliche Features und andere zusammenhängende Teile von Anwendungen gruppieren. Damit eine Komponente verwendet werden kann, muss sie immer in einem Modul deklariert werden – aber nur in genau einem.

Für viele Szenarien ist dieses Konzept genau passend und richtig – gelegentlich aber auch nicht: Besonders bei wiederverwendbaren Teilen, die in mehreren Modulen genutzt werden sollen, müssen wir uns häufig genauere Gedanken über die Struktur machen:

- Wo werden die Komponenten deklariert?
- Wo müssen welche Module importiert werden?
- Wie vermeide ich zirkuläre Referenzen?
- Wie behalte ich beim Refactoring alle Deklarationen im Blick?

Shared Module

Üblicherweise werden wiederverwendbare Teile in einem globalen Shared Module abgelegt, das überall dort importiert wird, wo eine dieser Komponenten benötigt wird. Dadurch entsteht leicht ein schwerfälliges und allwissendes Angular-Modul, das eine entkoppelte Struktur der Anwendung eher verhindert.

Bei der Planung der Architektur müssen wir das Konzept der Module immer berücksichtigen und im Hinterkopf behalten. Aus diesen Gründen herrschen in der Community seit jeher gemischte Gefühle über die Notwendigkeit von Angular-Modulen. Praktisch setzen manche Entwicklerinnen und Entwickler deshalb auf einen besonderen Ansatz, der auch als SCAM (Single-Component Angular Module) oder AIM (Angular Inline Module) bekannt ist: Jede Komponente erhält ein *eigenes* NgModule. Dadurch wird die Idee von Modulen, die *mehrere* zusammenhängende Bestandteile bündeln, fast vollständig aufgelöst. Eine Komponente muss in »ihr« Modul genau die Dinge importieren, die sie verwenden möchte – nicht mehr und nicht weniger. Die Idee ist eine Antwort auf die praktischen Unschönheiten, die mit NgModules auftreten können.

Single-Component Angular Module

Standalone Components

Nach mehr als fünf Jahren hat das Angular-Team den Wunsch nach weniger NgModules direkt adressiert: Das neue Feature heißt *Standalone Components* und wurde mit Angular 14 neu eingeführt. Eine Komponente, Pipe oder Direktive, die als standalone markiert ist, wird nicht in einem Modul deklariert, sondern wird alleinstehend verwendet. NgModules werden damit optional und müssen theoretisch gar nicht mehr verwendet werden. Die alleinstehenden Komponenten importieren selbst all die Dinge, die sie zum Funktionieren benötigen. Neben Services gehören dazu nun auch die Pipes, Direktiven und Kindkomponenten, die unsere Komponente in ihrem Template verwenden möchte.

Angular-Module und JavaScript-Module

Wenn wir in diesem Kapitel von »Modulen« sprechen, meinen wir stets Angular-Module mit dem Decorator `@NgModule()` – nicht aber die JavaScript-Module, mit denen wir die einfachen Imports und Exports organisieren.

25.2 Standalone Components erzeugen

Um eine Komponente, Pipe oder Direktive alleinstehend zu verwenden, setzen wir das Flag `standalone` im Decorator der Klasse:

Listing 25–1 Standalone Component definieren

```
@Component({
  selector: 'app-foo',
  standalone: true,
  // ...
})
export class FooComponent {}
```

Dadurch wird der Baustein unabhängig von einem Angular-Modul und kann alleinstehend genutzt werden. Für Pipes und Direktiven funktioniert dieser Weg genauso. Diese Einstellung können wir auch sofort beim Generieren mit der Angular CLI angeben:

```
$ ng g component foo --standalone
```

25.3 Abhängigkeiten definieren

NgModule: Kontext für Komponenten

Bisher haben wir eine Komponente stets in einem Modul deklariert. Das Modul definiert einen impliziten Kontext für alle enthaltenen Komponenten. Unter anderem legt es fest, welche anderen Komponenten, Pipes und Direktiven die Komponente in ihrem Template nutzen kann. Dieser »Befehlssatz« wird aus zwei Quellen zusammengesetzt:

- die Komponenten, Pipes und Direktiven, die direkt in dem Modul *deklariert* sind, und
- die exportierten Teile der anderen Module, die in das Modul *importiert* werden.

Zum Beispiel haben wir in Feature-Modulen immer das `CommonModule` importiert, um die eingebauten Pipes und Direktiven von Angular im Template nutzen zu können. Außerdem können wir alle anderen Komponenten, Pipes und Direktiven verwenden, die direkt unter `declarations` eingetragen wurden.

Kontext für Standalone Components

Mit diesem kleinen Ausflug in die Welt der Module kommen wir nun zurück zum Konzept der Standalone Components: Da sich eine Standalone Component nicht im Kontext eines Moduls befindet, müssen wir ihre Abhängigkeiten immer explizit angeben. Alles, was die Komponente in ihrem Template verwenden möchte, muss gezielt importiert werden. Dafür besitzt eine Standalone Component in ihren Metadaten die Eigenschaft `imports`.

Listing 25–2 Abhängigkeiten importieren

```
import { NgIf, NgFor, AsyncPipe, DatePipe } from '@angular/common';
// ...

@Component({
  // ...
  standalone: true,
  imports: [
    NgIf, NgFor,
    AsyncPipe, DatePipe,
    BarComponent, MyPipe
  ]
  // ...
})
export class FooComponent {}
```

Die eingebauten Direktiven und Pipes von Angular werden direkt aus dem jeweiligen Paket importiert, in der Regel aus `@angular/common`. Dabei müssen wir stets den Klassennamen der Direktive oder Pipe angeben. Unsere eigenen Komponenten etc., die wir im Template dieser Komponente verwenden wollen, importieren wir direkt aus dem lokalen Dateisystem. Beim Erstellen der Imports kann uns der Editor direkt aus dem Template heraus unterstützen.

Module importieren

Eine Standalone Component kann außerdem selbst Module importieren, deren Bestandteile sie in ihrem Template nutzen möchte. Wollen wir z. B. die Direktiven und Pipes von Angular nicht einzeln importieren, können wir das gesamte `CommonModule` einbinden. Dieser Weg ist vor allem für die Abwärtskompatibilität interessant: Nicht jede Drittanbieterbibliothek wird ihre Komponenten, Pipes und Direktiven als standalone anbieten, also können wir das gesamte NgModule direkt in der Komponente importieren:

Listing 25–3 Module importieren

```
@Component({
  // ...
  standalone: true,
  imports: [
    CommonModule,
    BooksSharedModule
  ]
})
export class FooComponent {}
```

Mehrere Teile bereitstellen

Um mehrere Komponenten, Pipes und Direktiven gemeinsam bereitzustellen, können diese auch als Array exportiert und importiert werden. Zum Beispiel kann eine Bibliothek all jene Direktiven zusammen ex-

portieren, die auch gemeinsam genutzt werden sollen. Auf diese Weise erhält man einen ähnlichen Komfort wie mit einem NgModule, das mehrere Dinge zur Nutzung bereitstellt. Diesen Ansatz können wir auch in unseren eigenen Anwendungen nutzen, um zusammenhängende Komponenten, Pipes und Direktiven in einem Rutsch zur Verfügung zu stellen.

Idealerweise sollte die Variable für ein solches Array passend nach ihrem Inhalt benannt werden, z. B. `AUTH_DIRECTIVES` oder `BOOK_PIPES`. Das haben wir im Codebeispiel mit `SHARED_THINGS` angedeutet.

Listing 25–4 *Array von Abhängigkeiten importieren*

```
// Array exportieren
export SHARED_THINGS = [BarComponent, MyPipe];


// in der Komponente
@Component({
  // ...
  standalone: true,
  imports: [NgIf, NgFor, SHARED_THINGS]
})
export class FooComponent {}
```

Zusammenfassung

Eine alleinstehende Komponente beschreibt also immer selbst ihre direkten Abhängigkeiten, indem die benötigten Teile importiert werden. Es gibt kein Modul, das den Kontext vorgibt. Dieser Ansatz sieht zunächst etwas aufwendiger aus, allerdings sind die tatsächlichen Beziehungen zwischen Komponenten so noch klarer erkennbar.

Eine Standalone Component darf übrigens nicht in den `declarations` eines Moduls eingetragen werden. Sie ist nur alleinstehend verwendbar und nicht von einem Modul abhängig.

25.4 Standalone Components in NgModules nutzen

Standalone Components können auch in einer modulbasierten Anwendung verwendet werden. Dazu können wir die Komponenten unter `imports` in einem NgModule eintragen. Wir behandeln die Komponente also so, als wäre sie ein Modul, dessen Bestandteile wir in unserem eigenen Modul verfügbar machen wollen. Im folgenden Beispiel wird die `FooComponent` importiert, also kann sie von den anderen Komponenten dieses Moduls verwendet werden.

Teil IV

Projektübergreifende Themen

26 Qualität fördern mit Softwaretests

»Program testing can be a very effective way
to show the presence of bugs,
but is hopelessly inadequate for showing their absence.«

Edsger W. Dijkstra
(Mathematiker, Informatiker und Wegbereiter
der strukturierten Programmierung)

26.1 Softwaretests

Wir, die Autoren dieses Buchs, lieben Softwaretests. Es geht uns darum, im hektischen Entwicklungsalltag einen kühlen Kopf zu bewahren und uns stets die notwendige Zeit für eine ordentliche Testabdeckung freizuhalten. Softwaretests geben uns ein gutes Gefühl. Wir wissen am Ende des Tages, dass wir einen guten Job gemacht haben, wenn alle Tests grün sind. Die Software ist dann zu einem hohen Grad fehlerfrei, sodass es später im Live-Betrieb keine bösen Überraschungen gibt. Das sorgt für zufriedene Nutzerinnen und Nutzer, ein gutes Karma und bedeutend mehr Spaß bei der Arbeit. Wer will schon Logfiles nach Feierabend auswerten und Bugs in Produktion analysieren? Wir nicht. Deshalb gehören Tests einfach dazu!

Tests stellen die Softwarequalität sicher.

Wenn wir in diesem Buch von Tests reden, so meinen wir immer *automatisierte Tests*. Wir werden manuelle Tests nicht betrachten – denn mit gutem Willen lässt sich so ziemlich alles automatisieren. Das Tooling rund um Angular hilft uns dabei.

Keine manuellen Tests

Beim Testing wollen wir beweisen, dass unsere Software die an sie gestellten Anforderungen fehlerfrei erfüllt. Weiterhin stellen Tests eine Dokumentation der fachlichen oder technischen Anforderungen dar. Zudem erhöhen Tests insgesamt die Qualität unserer Software, getesteter Code ist tendenziell modularer und lose gekoppelt – sonst wäre er nicht gut testbar. Eine gut gepflegte Sammlung an Tests bietet uns daher einen großen Mehrwert.

Dokumentation der Anforderungen

Abb. 26–1
Alles grün: Karma führt mehrere Unit-Tests erfolgreich aus.

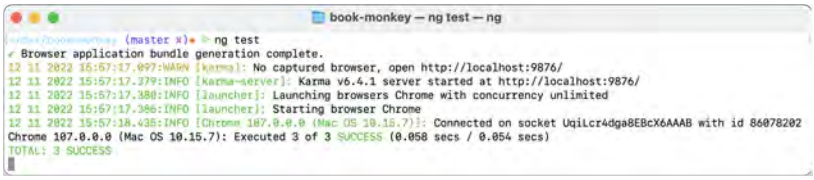
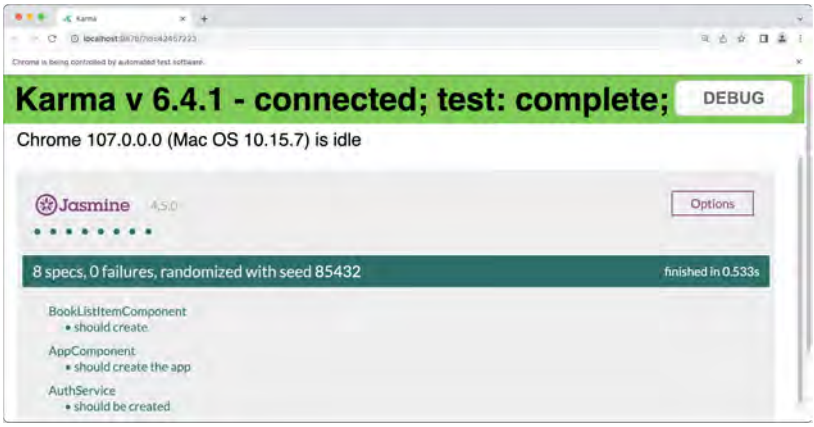


Abb. 26–2
Details zu den Karma-Tests im Chrome-Browser



26.2 Vorgehen beim Testing

Testabdeckung: Was sollte man testen?

Was man über einen Test spezifizieren sollte und *wie* man dies am besten tut – das zu beantworten ist eine der großen Herausforderungen beim Testing.

Lassen Sie uns ehrlich sein: Es ist praktisch unmöglich, im turbulenten Geschäftsalltag jedes technische Detail, jede Anforderung und jedes Akzeptanzkriterium durch einen Softwaretest abzubilden. Das muss allerdings auch gar nicht sein. Legen Sie für sich im Team fest, was der »Kern« der Anwendung ist: Wo darf man sich keinen Fehler erlauben? Wo sollte wirklich nichts schiefgehen? Was soll mein Test beweisen, damit die Anwendung fehlerfrei läuft?

100 % Testabdeckung sollten nicht das Ziel sein.

Uns ist folgender Fakt besonders wichtig: Sind die Ziele zu ehrgeizig, werden sie wahrscheinlich gar nicht erreicht. Manchmal hört man den Begriff *hundertprozentige Testabdeckung*. Das klingt zwar sehr erstrebenswert, aber Testabdeckung (*Code Coverage*) lenkt von den wichtigen Fragen ab. Die Testabdeckung ist eine Softwaremetrik, die wir aus unseren Unit- und Integrationstests ableiten können. Es wird ermittelt, wie viele Zeilen Code durch einen Test abgedeckt sind. Diese Metrik hilft uns dabei, weiße Flecken auf der Landkarte zu finden. Sie sagt hingegen nicht aus, ob der Test sinnvoll ist, ob alle notwendigen

Kombinationen getestet wurden und ob es überhaupt notwendig war, den abgedeckten Code zu testen. Das können nur Menschen entscheiden und keine Zahlen.

Testart: Wie sollte man testen?

Über die Art des Tests können wir bereits eine Reihe von generellen Eigenschaften ableiten. Deshalb wollen wir die verschiedenen Testarten kategorisieren.

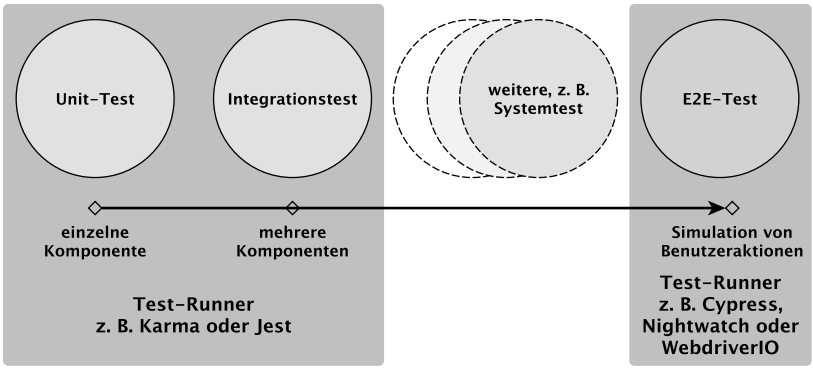


Abb. 26–3
Arten von Tests

Uns interessieren aus diesem Spektrum vor allem drei Arten von Tests:

- Unit-Tests
- Integrationstests
- End-to-end-Tests (kurz E2E, auch Oberflächentests genannt)

Unit-Tests

Unit-Tests überprüfen die kleinsten Einheiten (Units) einer Software. Diese Einheiten können unter anderem einzelne Methoden, Pipes, Services oder auch ganze Komponenten sein. Jeder andere Code, der nicht Teil der gerade getesteten Einheit ist, wird als *Abhängigkeit* bezeichnet. Bei einem Unit-Test ist es wichtig, dass wirklich nur eine einzige Einheit getestet wird. Das bedeutet, dass wir alle Abhängigkeiten durch sogenannte *Stubs* bzw. *Mocks* ersetzen sollten. Wir empfehlen Ihnen, so viele Tests wie möglich als Unit-Tests zu spezifizieren.

Integrationstests

In einer komplexen Webanwendung kann es manchmal sehr aufwendig sein, alle Abhängigkeiten sauber »auszumocken«. Es ist dann häufig einfacher, mehr als nur eine Einheit mit einem Test abzudecken. Sind mehrere Einheiten involviert, so nennt man dies einen *Integrationstest*. Es ist sinnvoll, einen Integrationstest zu schreiben, wenn der eingesparte Aufwand gegenüber einem Unit-Test entsprechend hoch ist! Nutzen und Kosten sind immer im Auge zu behalten. Spart man viel Zeit ein, kann man an anderer Stelle mehr testen. Weiterhin müssen wir auch in einigen Fällen unabwendbar Integrationstests schrei-

ben. Es kann nämlich passieren, dass zwei Units zwar isoliert betrachtet funktionieren, sie dies aber im Zusammenspiel nicht mehr tun. Wenn man eine solche Situation erkennt, sollte man das Zusammenspiel mehrerer Einheiten mit einem Integrationstest sicherstellen.

Systemtest

Es gibt je nach Definition weitere Teststufen. Ein *Systemtest* beinhaltet beispielsweise die Ausführung von Tests über das gesamte System hinweg. Mit den vorgestellten Tools lassen sich auch solche Tests je nach Situation automatisiert abbilden, allerdings sind die Anforderungen an einen Systemtest sehr produktspezifisch. Wir werden in diesem Buch nicht näher darauf eingehen, da wir den Fokus auf Themen rund um Angular richten wollen.

Oberflächentests

Oberflächentests ergänzen unsere Unit- und Integrationstests. Wie der Name vermuten lässt, wird mit diesen Tests die grafische Benutzeroberfläche einer Anwendung getestet. Ein echter Browser wie Chrome oder Firefox wird dabei ferngesteuert und besucht eine vollständige Website. Unit- und Integrationstests eignen sich gut dafür, einzelne Anforderungen aus der technischen Sicht zu beschreiben. Oberflächentests können hingegen besonders gut komplette fachliche Funktionen der Anwendung spezifizieren. So lässt sich durch E2E-Tests leichter die Perspektive der Endanwenderinnen und -anwender einnehmen.

26.3 Test-Framework Jasmine

Wir kennen nun eine grobe Unterteilung von Testarten und wollen zum Einstieg unseren ersten Unit-Test schreiben. Bevor es aber tatsächlich mit dem Testing losgehen kann, müssen wir noch ein Test-Framework wählen. Ein solches Framework bietet uns ein technisches Grundgerüst für die Definition von Tests. Es ist dabei egal, ob wir einen Unit-Test, Integrationstest oder Oberflächentest schreiben – ein Test-Framework benötigt man immer. Eines der bekanntesten Frameworks für JavaScript-Tests ist *Jasmine*. Die Angular CLI hat bereits Jasmine installiert und konfiguriert, wir können also sofort loslegen.

Behavior Driven Development

Jasmine bringt eine Syntax im Stil des Behavior Driven Development (BDD) mit. Man schreibt dabei die Tests in natürlicher Sprache auf. Dies geschieht durch einfache Strings, die später im Test-Runner sichtbar ausgegeben werden:

Listing 26–1 *Test in natürlicher Sprache*

```
describe('Deep Thought', () => {
  it('should know the answer to life, the universe and
    ↪ everything', () => {

  });
});
```

Im Output liest man später den vollständigen Satz:

```
DeepThought should know the answer to life,
the universe and everything
```

Wir erhalten einen gut lesbaren Output mit ganzen Sätzen, sodass wir bereits einen Teil der Dokumentation kennen, ohne den Quelltext betrachtet haben zu müssen. So wollen wir alle unsere Tests definieren – als verständliche Sätze, die sich später wie ein Handbuch lesen lassen!

Tests als verständliche Sätze

Noch hat die gezeigte leere Testhülle keinerlei Funktionalität. Wir füllen sie daher mit einem ersten Beispiel:

Listing 26–2
Ein erster Unit-Test mit Jasmine

```
export class DeepThought {
  getAlmightyAnswer() {
    return 42;
  }
}

describe('Deep Thought', () => {
  let deepThought: DeepThought;

  beforeEach(() => {
    // Arrange
    deepThought = new DeepThought();
  });

  it('should know the answer to life, the universe and
    ↪ everything', () => {
    // Act
    const answer = deepThought.getAlmightyAnswer();

    // Assert
    expect(answer).toBeGreaterThan(0);
  });
});
```

Wir sehen im Beispiel mehrere Funktionen, die von Jasmine bereitgestellt werden:

Funktionen von Jasmine

- `describe()`
- `it()`
- `expect()`
- `beforeEach()`

Die Funktion `describe()` definiert eine Sammlung (»test suite«) zusammenhängender Spezifikationen bzw. Tests. Der Aufruf erwartet zwei Argumente: Das erste Argument ist ein String und beschreibt als Wort oder mit mehreren Worten, welche Sache gerade getestet wird. Das zweite Argument ist eine Funktion, die alle Spezifikationen (»specs«) beinhaltet. `describe()`-Blöcke können beliebig tief verschachtelt werden, um die Übersichtlichkeit zu erhöhen und um redundanten Code zu vermeiden.

it()

Die Funktion `it()` steht wiederum für eine einzelne Spezifikation. Auch eine Spezifikation wird mit mehreren Worten oder gerne auch mit einem ganzen Satz beschrieben. Eine Spezifikation enthält eine oder mehrere Erwartungen (`expect()`), die zur Laufzeit geprüft werden und die für einen erfolgreichen Durchlauf erfüllt sein müssen. Die Funktionen `beforeEach()` bzw. `afterEach()` laufen, wie der Name vermuten lässt, stets vor bzw. nach jeder Spezifikation ab. Setzt man `describe()` und `beforeEach()` geschickt ein, so lässt sich viel redundanter Code bei der Initialisierung vermeiden.

expect()
beforeEach()
afterEach()

Tab. 26–1 *Die wichtigsten Funktionen von Jasmine*

Funktion	Beschreibung
`describe(description: string, specDefinitions: () => void)`	definiert eine Sammlung von Spezifikationen (»test suite«)
`beforeAll(action: () => void)`	wird nur einmal *vor* allen Spezifikationen ausgeführt
`beforeEach(action: () => void)`	wird *vor* jeder Spezifikation ausgeführt
`it(expectation: string, assertion: () => void)`	Spezifikation (»spec«), unpräzise ausgedrückt auch einfach nur »Test«
`expect(actual: any)`	Erwartung, wird zusammen mit einem Matcher (z. B. `toBe()`) verwendet
`afterEach(action: () => void)`	wird *nach* jeder Spezifikation ausgeführt
`afterAll(action: () => void)`	wird nur einmal *nach* allen Spezifikationen ausgeführt

Jasmine Matchers

Eine Erwartung wird immer mit einem Matcher kombiniert. So prüft man etwa mit `expect(1).toBe(1)` bzw. mit `expect(1).not.toBe(2)` auf strikte Gleichheit.

Die wichtigsten Matchers von Jasmine finden Sie im Anhang ab Seite 871 aufgelistet. Eine ausführliche Liste ist auf der Homepage von Jasmine zu finden.[1] Dank der Typdefinitionen von TypeScript erhalten

[1] https://ng-buch.de/c/65 – Jasmine: Included Matchers

27 Barrierefreiheit (a11y)

»Barrierefreiheit ist für zehn Prozent der Menschen essenziell, für 40 Prozent notwendig und für 100 Prozent komfortabel. Aus diesem Grund sollte die digitale Barrierefreiheit als festes Qualitätsmerkmal ihren Platz in der Softwareentwicklung finden.«

Mohammed Malekzadeh
(Experte für digitale Barrierefreiheit bei der Deutschen Bahn AG)

Das World Wide Web ist ein global genutztes Medium: Menschen aus verschiedenen Kulturen, mit diversen Lebenssituationen und vielfältigen Hintergründen haben Zugang zum Internet. Während noch vor einigen Jahren viele Vorgänge in Papierform abgewickelt wurden, werden diese nun zunehmend digitalisiert. Das hat Vorteile: schnellere Abwicklung, Automatisierung und Ressourceneinsparungen bei Verbrauchsmaterialien. Die Entwicklung bringt aber auch Herausforderungen mit sich, die nicht immer auf den ersten Blick deutlich sind: Unsere Software soll von möglichst jedem einzelnen Menschen bedienbar sein. Im Umkehrschluss müssen wir also Wege finden, wie Menschen mit Einschränkungen die Anwendung nutzen können. Diese Einschränkungen können sehr vielfältig sein – von visuellen und akustischen bis hin zu motorischen oder kognitiven Beeinträchtigungen.

Inklusion aller Menschen

Digitale Barrierefreiheit (*engl. Accessibility (a11y)*[1]) hat zum Ziel, die Barrieren zur unkomplizierten Benutzung der Anwendung zu verringern und so möglichst jede einzelne Person zu inkludieren. Das ist nicht immer einfach, und faktisch gibt es keine digitale Anwendung, die absolut keine Barrieren hat. Das liegt schon allein daran, dass die Zugriffsmöglichkeiten heutzutage divers sind und nicht alle bedacht und getestet werden können. Dennoch lassen sich die Grundlagen optimieren, und es sollte das Ziel sein, so viele Barrieren wie möglich abzuschaffen, um eine Vielzahl von Anwendungsfällen für Menschen mit Beeinträchtigungen zu berücksichtigen. Sprechen wir also von Barriere-

Barrierearme Anwendung

[1] Die Abkürzung *a11y* ist ein *Numeronym*. Die Zahl *11* steht für die Anzahl der Buchstaben zwischen dem ersten und letzten Buchstaben des Wortes *accessibility*.

freiheit, geht es immer darum, eine Anwendung zu einem hohen Grad von Barrieren zu befreien.

Barrieren gibt es für jeden.

Dabei geht es nicht nur um dauerhafte körperliche oder geistige Beeinträchtigungen: Stellen Sie sich nur einmal vor, Sie brechen sich die Hand bei einem Unfall und können für mehrere Monate die Maus am Computer nicht bewegen. Sie werden sich wünschen, dass die Software, die Sie nutzen, barrierefrei ist, indem sie z. B. durch Tastatureingaben oder Sprachbefehle bedient werden kann.

Vor diesem Hintergrund können wir Einschränkungen grundsätzlich in drei Gruppen untergliedern:

- **permanent:** z. B. Menschen mit Seh- oder Gehöreinschränkungen
- **temporär:** z. B. Personen mit einer Ohrentzündung oder einem gebrochenen Arm
- **situationsbedingt:** z. B. Eltern, die ihr Kind im Arm halten, oder Menschen, die gerade ein Auto steuern

Abwägungen und Kompromisse

Diese Gruppierung lässt sich auch auf die allgemeine User Experience übertragen. Grundsätzlich gibt es beim Thema Barrierefreiheit kein binäres *Schwarz und Weiß*: So sollte die Schrift auf der Seite z. B. nicht *grundsätzlich* größer als üblich formatiert werden. Menschen ohne Seheinschränkung könnten diese Optimierung als störend empfinden, da sie spezifisch für eine bestimmte Zielgruppe ist. Als Mensch im sehr gehobenen Alter haben Sie möglicherweise erst vor kurzer Zeit den Umgang mit einem Computer gelernt. Sie wollen nun auf einer Website eine Bestellung vornehmen. Wahrscheinlich werden Sie mit einem sehr einfachen Step-by-Step-Formular mit vielen Hilfen und Erläuterungen sehr gut zurechtkommen. Was für eine Person in dieser Situation super ist, kann bei erfahrenen Menschen, die täglich solche Bestellungen ausführen, zu Frustration führen, weil die Anzahl der vielen Klicks und Hilfen zeitraubend wirkt. Ein einzelnes Formular, das auf Effizienz optimiert ist, ist vermutlich die bessere Lösung. Und hier wird es schwierig: Wir müssen unsere Zielgruppe genau kennen und Kompromisse finden. Gegebenenfalls müssen wir Mehraufwand in Kauf nehmen, um verschiedene Ansätze für verschiedene Zielgruppen zu unterstützen. Unsere Empfehlung: Analysieren Sie die Zielgruppe genau, holen Sie sich Unterstützung und Expertise aus dem Bereich der UX und personifizieren Sie Ihre Anwenderinnen und Anwender. Setzen Sie außerdem auf barrierefreie Komponentenbibliotheken und machen Sie sich Gedanken bei der Auswahl der richtigen semantischen HTML-Elemente. Mit einfachen Mitteln können Sie von Beginn an immer wieder wichtige Aspekte der Barrierefreiheit prüfen, um später aufwendiges Refactoring und Anpassungen von Details zu vermeiden.

Barrierefreiheit von Anfang an

Wir möchten in diesem Kapitel ein paar Tipps geben, um Angular-Anwendungen mit technischen Maßnahmen möglichst barrierefrei umzusetzen. Dabei gehen wir nach einem kurzen Blick auf die zugrunde liegenden Gesetze und Standards auf vier große Themen ein:

- Generelle Aspekte, unabhängig von Angular
- Features von Angular
- ESLint-Regeln für Angular
- Angular Component Development Kit (CDK)

Außerdem möchten wir Ihnen einige Tools an die Hand geben, mit denen Sie bestimmte Aspekte der Barrierefreiheit überprüfen und messen können.

27.1 Gesetze und Standards

Die Umsetzung digitaler Barrierefreiheit ist in verschiedenen Standards und Richtlinien geregelt, unter anderem in gesetzlichen Regelungen. Für verschiedene Anwendungstypen muss also ein gewisses Maß an Barrierefreiheit per Gesetz umgesetzt werden. Wir wollen in diesem Abschnitt kurz auf diese grundlegenden Richtlinien und geltenden Gesetze in Deutschland und Europa eingehen.

Gesetzliche Definition

Barrierefreiheit ist ein gesetzlich festgelegter Begriff. In Deutschland wird die Barrierefreiheit im § 4 Behindertengleichstellungsgesetz (BGG) wie folgt definiert:[2]

> *Barrierefrei sind bauliche und sonstige Anlagen, Verkehrsmittel, technische Gebrauchsgegenstände, Systeme der Informationsverarbeitung, akustische und visuelle Informationsquellen und Kommunikationseinrichtungen sowie andere gestaltete Lebensbereiche, wenn sie für Menschen mit Behinderungen in der allgemein üblichen Weise, ohne besondere Erschwernis und grundsätzlich ohne fremde Hilfe auffindbar, zugänglich und nutzbar sind. Hierbei ist die Nutzung behinderungsbedingt notwendiger Hilfsmittel zulässig.*

[2] https://ng-buch.de/c/76 – Bundesamt für Justiz – § 4 BGG

Aus dieser Definition können wir die folgenden Punkte ableiten:

- Unsere Produkte sollen in der allgemein üblichen Weise für Menschen mit Einschränkungen zugänglich sein. Sonderlösungen sollten, sofern möglich, vermieden werden. Das gleiche Produkt sollte also in seiner Originalform von jedem Menschen gut verwendbar sein.
- Die Nutzung sollte realistischerweise durch Menschen mit durchschnittlichen Kenntnissen erfolgen können.
- Der Weg bis zum Produkt muss ebenfalls barrierefrei sein. Auf die digitale Welt bezogen bedeutet das, dass es nicht ausreicht, lediglich ein barrierefreies Dokument oder eine Webseite zur Verfügung zu stellen. Auch die Navigation dorthin muss barrierefrei gestaltet werden.
- Es sollte sichergestellt werden, dass Produkte mit gängigen Hilfsmitteln (z. B. Screenreader für Menschen mit Blindheit und Sehbehinderung) konsumierbar sind.

Im Rahmen dieses Buchs schlagen wir die folgende verkürzte und allgemeinere Definition für die Barrierefreiheit vor:

> *Barrierefrei ist ein Angebot dann, wenn es von jedem Menschen unter Zuhilfenahme seiner notwendigen Hilfsmittel genutzt werden kann.*

Wir weiten die Definition der Barrierefreiheit also auf *alle Menschen* aus, weil wir der Überzeugung sind, dass jeder Mensch von der Barrierefreiheit profitiert.

WAI und WCAG

Das World Wide Web Consortium (W3C) arbeitet kontinuierlich an den technischen Standards, die im WWW benötigt werden. Innerhalb des W3C gibt es eine Gruppe, die sich ausschließlich mit dem Thema Barrierefreiheit im Kontext des Webs beschäftigt – die Web Accessibility Initiative (WAI). Das Resultat ihrer Arbeit sind die Web Content Accessibility Guidelines (WCAG), also Richtlinien für barrierefreie Webinhalte.[3] Sie sind in vier wesentliche Prinzipien untergliedert:

1. **wahrnehmbar:** Informationen und Bestandteile müssen so präsentiert werden, dass alle sie wahrnehmen können.

[3] https://ng-buch.de/c/77 – WCAG 2 Overview | Web Accessibility Initiative (WAI) | W3C

Teil VI

Fortgeschrittene Themen

32 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 unsere Anwendung bisher stets 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 vergleichsweise klein und übersichtlich – in der Praxis werden die Anwendungen 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 »Hüterinnen« 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 ist ein entscheidender Unterschied zum bisherigen Vorgehen, 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.

32.1 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 32–1
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()` manipuliert werden kann. 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 32–2
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 32–3 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. Eine Änderung am State wird zwar jederzeit korrekt angezeigt, aber dies basiert allein auf den Mechanismen der Change Detection.[2] 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.[3] 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] Zur Funktionsweise und Optimierung der Change Detection in Angular haben wir unter »Wissenswertes« ab Seite 805 einen Abschnitt untergebracht.

[3] Im Kapitel zur Reaktiven Programmierung mit RxJS haben wir Subjects ausführlich besprochen, siehe Seite 261.

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 32–4
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 32–5
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 32–6
Ergebnis mit der AsyncPipe anzeigen

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

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

Zusammenfassung aller Konzepte

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

Redux

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

32.2 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 eigenen Account – 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*[9] und *read-only*. Wir können die Daten aus dem State nicht verändern, sondern ausschließlich lesend darauf zugreifen. Möch-

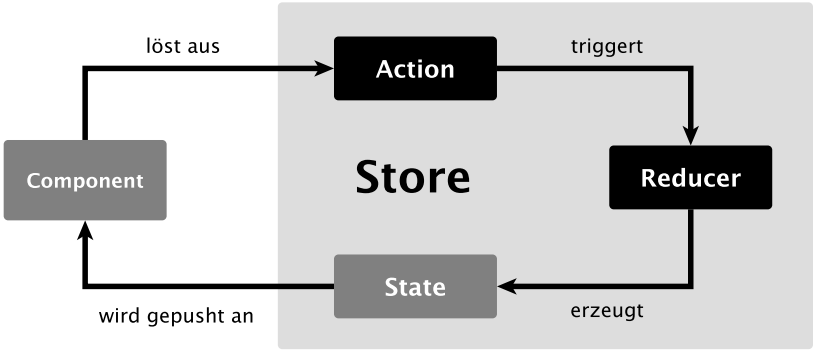
[9] Das State-Objekt ist theoretisch mutierbar, wir werden es aber stets immutable behandeln. NgRx setzt also auf die Pseudo-Immutability.

ten 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 *Reducers* 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 32–1 grafisch dargestellt. Hier ist gut erkennbar, dass die Daten stets in eine Richtung fließen und dass Lesen und Schreiben klar voneinander getrennt sind.

Abb. 32–1
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 Reducers 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 existie-

ren 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
- Elf

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 Elf zu werfen.

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]

32.3 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 genauer besprechen. Außerdem wollen wir im BookMonkey einen Aspekt mithilfe von NgRx umsetzen, um so alle Bausteine auch praktisch zu üben.

[10] https://ng-buch.de/c/131 – Ordina JWorks Tech Blog: NGRX vs. NGXS vs. Akita vs. RxJS: Fight!

32.3.1 Projekt vorbereiten

Als Grundlage für diesen praktischen Teil verwenden wir das Beispielprojekt BookMonkey in der Version nach dem Kapitel zu Guards, also bevor wir die Anwendung auf Standalone Components umgestellt haben. Möchten Sie mitentwickeln, so können Sie Ihr bestehendes BookMonkey-Projekt verwenden oder neu starten und den Code über GitHub herunterladen:

Listing 32–14 *BookMonkey als Grundlage für NgRx verwenden*

```
$ git clone https://ng-buch.de/bm5-16-guards.git book-monkey-ngrx
$ cd book-monkey-ngrx
$ npm install
```

32.3.2 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 --defaults
$ 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 später in Abschnitt 32.4 ab Seite 717 genauer darauf eingehen, um den Lesefluss in diesem Kapitel nicht zu unterbrechen.

32.3.3 Schematics nutzen

Um nach der Einrichtung die 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, Reducers und Effects bequem mithilfe von `ng generate` anlegen können. Auch diese Abhängigkeit wird mittels `ng add` installiert.

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

Schematic Collection festlegen

Die Schematics von NgRx werden durch diesen Aufruf automatisch im Projekt registriert. Jeder Aufruf von `ng generate` durchsucht dann

auch die Skripte in diesem Paket. So können wir bequem einen Befehl wie `ng generate action` verwenden, ohne die Zielkollektion explizit angeben zu müssen. Die Collection wird mit einem Eintrag in der Datei `angular.json` festgelegt, den Sie jederzeit wieder löschen oder ändern können, falls Sie die Skripte von NgRx nicht mehr nutzen möchten.

32.3.4 Grundstruktur

Die ausgeführten Befehle haben bereits alles Nötige eingerichtet, sodass wir sofort mit der Implementierung beginnen können. Vorher wollen wir jedoch einen Blick auf die Änderungen werfen, die von den Schematics an unserem Projekt vorgenommen wurden.

Neben allen benötigten Abhängigkeiten in der `package.json` sind einige neue Imports im `AppModule` hinzugekommen. Das `StoreModule` bringt den Kern des NgRx-Stores in die Anwendung. Die verwendete Methode `forRoot()` erwartet zwei Argumente: Im ersten Objekt können wir angeben, welche Reducers für welchen Teil des State-Objekts verantwortlich sind (eine sogenannte `ActionReducerMap`). Üblicherweise nutzen wir dieses Objekt nicht, um die State-Struktur für unsere Features zu definieren, denn dafür existiert ein anderer, dynamischerer Weg. Verwenden wir allerdings das Paket `@ngrx/router-store`, so müssen wir das Mapping für den Router hier statisch im `AppModule` konfigurieren.[11] Im zweiten Argument von `forRoot()` können wir ein Konfigurationsobjekt übergeben. Da wir diese beiden Aspekte momentan nicht nutzen wollen, sind lediglich zwei leere Objekte als Argumente angegeben.

ActionReducerMap

Konfiguration für den Store

Außerdem sind zwei weitere Imports für `EffectsModule` und `StoreDevToolsModule` eingetragen worden. Diese beiden Module binden Effects und die Store DevTools ein, wir wollen uns aber zu diesem Zeitpunkt noch nicht detaillierter damit auseinandersetzen.

Listing 32–15 *Imports für NgRx im* AppModule (app.module.ts)

```
// ...
import { NgModule, isDevMode } from '@angular/core';
import { StoreModule } from '@ngrx/store';
import { StoreDevtoolsModule } from '@ngrx/store-devtools';
import { EffectsModule } from '@ngrx/effects';

@NgModule({
  // ...
  imports: [
    // ...
```

[11] Auf das Paket `@ngrx/router-store` gehen wir zum Ende dieses Kapitels ab Seite 721 noch ein.

Teil VII

Wissenswertes

36 Fortgeschrittene Konzepte für Komponenten

Komponenten sind die wichtigsten Bausteine einer jeden Anwendung. Mit all ihren Details sind sie deshalb ein sehr umfangreiches Feld, in dem es auch für erfahrene Personen stets neue Dinge zu entdecken gibt. Wir haben im Verlauf dieses Buchs bereits viele Facetten von Komponenten und Templates kennengelernt: die Template-Syntax, Bindings, Direktiven, Lifecycle Hooks, den `<ng-container>` und vieles mehr. In diesem Kapitel möchten wir Ihnen einige fortgeschrittene Aspekte vorstellen, die für die Entwicklung von Komponenten interessant sind.

36.1 Else-Block für die Direktive `ngIf`

Die Strukturdirektive `ngIf` sorgt dafür, dass Teile des Templates abhängig von einer Bedingung angezeigt werden. Dabei werden die betreffenden Elemente des DOM nicht einfach aus- oder eingeblendet, sondern komplett entfernt bzw. wieder hinzugefügt. Listing 36–1 zeigt den einfachsten Einsatz der Direktive. Es wird lediglich überprüft, ob der Wert des Propertys `show` wahr ist. Im positiven Fall wird das Element dem DOM hinzugefügt und gerendert.

```
<button (click)="show = !show">Toggle</button>

<p *ngIf="show">
  Bedingung wahr, Text wird angezeigt.
</p>
```

Listing 36–1
nglf *nutzen*

Die Schreibweise mit dem Stern (`*ngIf`) ist dabei nur eine Kurzform. Intern wird sie zur folgenden längeren Variante aufgelöst, die ebenso gültig ist:

```
<ng-template [ngIf]="show">
  <p>Bedingung wahr, Text wird angezeigt.</p>
</ng-template>
```

Listing 36–2
nglf
in Langschreibweise mit <ng-template>

Angular nutzt unter der Haube das Element `<ng-template>`. Ist die Bedingung der Direktive wahr, so wird der Inhalt des Templates an dieser Stelle in den DOM eingebaut. Um das Verhalten im Detail zu verstehen, sollten Sie sich den Abschnitt zu Strukturdirektiven ab Seite 445 durchlesen. Dort erfahren Sie mehr über das Verhalten von Strukturdirektiven und welche verschiedenen Möglichkeiten Sie für die Umsetzung haben. Zunächst reicht es aber auch, wenn Sie wissen, dass das Template-Element normalerweise nicht in den DOM übernommen wird und erst eine Strukturdirektive dafür sorgt, dass es dargestellt wird.

Alternatives Template einblenden

Die Direktive `ngIf` besitzt einen optionalen `else`-Block. Er kann genutzt werden, um ein alternatives Template einzublenden, wenn die angegebene Bedingung in `ngIf` nicht erfüllt ist. Dabei muss das alternative Template in einem `<ng-template>` definiert werden. Es wird über eine lokale Elementreferenz adressiert; im folgenden Beispiel haben wir dafür den Namen `elseTempl` verwendet. Anschließend wird im `ngIf` der `else`-Zweig mit diesem Template verknüpft: `else elseTempl`. Ist die Prüfbedingung falsch, so wird der Inhalt des angegebenen Templates geladen und dargestellt. `show` sei im folgenden Beispiel ein Property in der Komponente vom Typ `boolean`.

Listing 36–3 nglf *mit* else-*Block nutzen*

```
<p *ngIf="show; else elseTempl">
  Bedingung wahr
</p>

<ng-template #elseTempl>
  <p>Bedingung unwahr</p>
</ng-template>
```

36.2 TrackBy-Funktion für die Direktive `ngFor`

Die Direktive `ngFor` iteriert über ein Array und erzeugt für jedes Array-Element ein neues DOM-Element. Oft besteht das Array nicht aus einzelnen Literalwerten, sondern wir iterieren über ein Array von Objekten. Tauschen wir ein Objekt des Arrays aus, z. B. weil wir neue Daten erhalten, so wird das zugehörige DOM-Element zerstört und anschließend erneut mit neuen Daten hinzugefügt. Das Entfernen des DOM-Elements und das erneute Hinzufügen sind aufwendig. Bei Arrays mit nur einer geringen Anzahl von Elementen werden Sie diesen Effekt kaum spüren. Verarbeiten Sie jedoch ein größeres Array, so kann die Performance der Anwendung darunter leiden.

DOM-Elemente werden komplett neu erzeugt.

Usability und Barrierefreiheit

Ein weiteres Problem entsteht im Zusammenhang mit der Barrierefreiheit und Usability. Setzen wir den Fokus auf ein Element, das von `ngFor` erstellt wurde, so verliert das Element den Fokus, sobald es neu

gerendert wird. Das macht sich besonders bemerkbar, wenn wir Formularfelder mit ngFor erstellen: Ändern sich die Daten im Array, wird die View neu gerendert, und das Formularfeld verliert den Fokus. Besonders schwer ins Gewicht fällt diese Eigenheit, wenn man die Seite mit einem Screenreader betrachtet. Der Reader kann ein erneuertes Element nicht mehr verfolgen und springt an eine andere Stelle.

Referenzvergleich

Das Problem kommt daher, dass Objekte und Arrays in JavaScript stets nur als Referenz gespeichert werden. Angular kann also die inhaltliche Gleichheit von zwei Objekten nicht feststellen: Haben zwei Objekte denselben Inhalt, aber unterschiedliche Speicherstellen, so gelten sie als unterschiedlich – ngFor rendert den DOM also neu, wenn eine solche Änderung eintritt. Bei Literalen wie Strings oder Zahlen ist das kein Problem, denn hier wird der tatsächliche Wert verglichen.

trackBy *verwenden*

Um diesen Problematiken entgegenzuwirken, können wir auf der Direktive ngFor eine sogenannte trackBy-Funktion nutzen. Sie legt fest, nach welchem Merkmal ein Objekt identifiziert wird. Damit kann Angular die inhaltliche Identität von Objekten feststellen und verhindert das ständige Neuerzeugen der DOM-Elemente. trackBy wird an den Ausdruck im ngFor angehängt. Wir teilen der Option mit, welche Methode zum Tracken der Elemente genutzt werden soll. Der Bezeichner trackBook verweist also auf die Methode trackBook() aus der Komponentenklasse:

Listing 36–4
trackBy *auf der Direktive* ngFor *nutzen*

```
<span *ngFor="let book of books; trackBy: trackBook">
  {{ book.isbn }} / {{ book.title }}
</span>
```

trackBook() ist eine Methode mit zwei Argumenten: Das erste Argument beinhaltet immer den aktuellen Iterationsindex des Arrays. Als zweites wird das iterierte Array-Element übergeben. Die Methode muss nun einen eindeutigen Schlüssel für jedes Array-Element zurückgeben. Wenn das Array also Bücher beinhaltet, ist der eindeutige Schlüssel die ISBN. Sollte unser Array keinen solchen Schlüssel besitzen, kann auch alternativ der index zurückgeliefert werden.

Listing 36–5
TrackBy-Funktion definieren

```
// ...
trackBook(index: number, book: Book) {
  console.log('TrackBy:', book.isbn, 'index:', index);
  return book.isbn;
  // alternativ: return index;
}
```

Um bei der Typisierung keinen Fehler zu machen, bietet Angular den Typ TrackByFunction<T> an. Die Funktion müssen wir in diesem Fall als Arrow Function in einem Property der Klasse ablegen. Die Typen für die Argumente der Funktion werden dann automatisch ermittelt.

Listing 36–6 *TrackBy-Funktion typisieren*

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

// ...
trackBook: TrackByFunction<Book> = (index, book) => {
  return book.isbn;
}
```

trackBy *hält das DOM-Element.*

Die Direktive `ngFor` arbeitet nun mit `trackBy` und hält das DOM-Element, anstatt es zu entfernen und neu zu erzeugen. Bei Änderungen an den Daten werden nur die einzelnen Bindings innerhalb des DOM-Elements aktualisiert. Das ist im Gegensatz zur Neuerzeugung wesentlich ressourcensparender. Außerdem geht z. B. bei Eingabefeldern der Fokus nicht mehr verloren, und wir können durchgehend im Eingabefeld weitertippen, auch wenn die Objekte im Array währenddessen ausgetauscht werden.

Um die Problematik und den Vergleich zur Arbeitsweise mit und ohne `trackBy` besser zu veranschaulichen, haben wir ein StackBlitz-Projekt bereitgestellt:

Demo und Quelltext:
https://ng-buch.de/c/stackblitz-trackby

36.3 Container und Presentational Components

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

Index