

O'REILLY®

3. Auflage

# Angular

Das Praxisbuch zu Grundlagen  
und Best Practices



Manfred Steyer

Papier  
plus<sup>+</sup>  
PDF.

Zu diesem Buch – sowie zu vielen weiteren O'Reilly-Büchern – können Sie auch das entsprechende E-Book im PDF-Format herunterladen. Werden Sie dazu einfach Mitglied bei oreilly.plus+:  
[www.oreilly.plus](http://www.oreilly.plus)

**3. AUFLAGE**

---

# **Angular**

*Das Praxisbuch zu Grundlagen und Best Practices*

*Manfred Steyer*

**O'REILLY®**

Manfred Steyer

Lektorat: Ariane Hesse

Fachliche Unterstützung: Hans-Peter Grahs

Korrektorat: Sibylle Feldmann, [www.richtiger-text.de](http://www.richtiger-text.de)

Satz: III-satz, [www.drei-satz.de](http://www.drei-satz.de)

Herstellung: Stefanie Weidner

Umschlaggestaltung: Michael Oréal, [www.oreal.de](http://www.oreal.de)

Druck und Bindung: mediaprint solutions GmbH, 33100 Paderborn

Bibliografische Information der Deutschen Nationalbibliothek

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

ISBN:

Print 978-3-96009-166-0

PDF 978-3-96010-576-3

ePub 978-3-96010-577-0

mobi 978-3-96010-578-7

3. Auflage 2021

Copyright © 2021 dpunkt.verlag GmbH

Wieblinger Weg 17

69123 Heidelberg

Dieses Buch erscheint in Kooperation mit O'Reilly Media, Inc. unter dem Imprint »O'REILLY«.

O'REILLY ist ein Markenzeichen und eine eingetragene Marke von O'Reilly Media, Inc. und wird mit Einwilligung des Eigentümers verwendet.

#### Hinweis:

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



#### Schreiben Sie uns:

Falls Sie Anregungen, Wünsche und Kommentare haben, lassen Sie es uns wissen: [kommentar@oreilly.de](mailto:kommentar@oreilly.de).

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Die Verwendung der Texte und Abbildungen, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags urheberrechtswidrig und daher strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung oder die Verwendung in elektronischen Systemen.

Es wird darauf hingewiesen, dass die im Buch verwendeten Soft- und Hardware-Bezeichnungen sowie Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen.

Alle Angaben und Programme in diesem Buch wurden mit größter Sorgfalt kontrolliert. Weder Autor noch Verlag können jedoch für Schäden haftbar gemacht werden, die in Zusammenhang mit der Verwendung dieses Buches stehen.

---

# Inhalt

<b>Vorwort .....</b>	<b>15</b>
<b>1 Projekt-Setup .....</b>	<b>21</b>
Visual Studio Code .....	21
Angular CLI .....	23
Node.js und Angular CLI installieren .....	23
Ein Projekt mit der CLI generieren .....	23
Angular-Anwendung starten .....	24
Build mit CLI .....	26
Projektstruktur von CLI-Projekten .....	27
Internet Explorer 11 .....	31
Eine Style-Bibliothek installieren .....	32
Alternativen zu Bootstrap .....	34
Zusammenfassung .....	35
<b>2 Erste Schritte mit TypeScript .....</b>	<b>37</b>
Motivation .....	37
Mit TypeScript starten .....	38
Hallo Welt! .....	38
Variablen deklarieren .....	39
Ausgewählte Datentypen in TypeScript .....	41
Ein erstes Objekt samt Modul .....	44
Auto-Importe mit Visual Studio Code .....	47
Klassen .....	47
Funktionen und Lambda-Ausdrücke .....	50
Interfaces und Vererbung .....	53
Interfaces .....	53
Klassenvererbung .....	56
Type Assertion (Type Casting) .....	58

Abstrakte Klassen . . . . .	59
Zugriff auf die Basisklasse . . . . .	60
Ausgewählte Sprachmerkmale . . . . .	61
Getter und Setter . . . . .	61
Generics . . . . .	62
Exceptions . . . . .	64
Spread-Operator . . . . .	66
Strikte Null-Prüfungen . . . . .	66
Asynchrone Operationen . . . . .	68
Callbacks und die Pyramide of Doom . . . . .	69
Promises . . . . .	70
async und await . . . . .	71
Bedeutung von Promises in Angular . . . . .	72
Zusammenfassung . . . . .	72
<b>3 Eine erste Angular-Anwendung . . . . .</b>	<b>73</b>
Angular-Komponente erzeugen . . . . .	74
Komponentenlogik . . . . .	75
Auf das Backend zugreifen . . . . .	77
Templates und die Datenbindung . . . . .	82
Two-Way-Binding . . . . .	82
Property-Bindings . . . . .	84
Direktiven . . . . .	84
Pipes . . . . .	86
Event-Bindings . . . . .	86
Das gesamte Template . . . . .	87
Komponenten einbinden . . . . .	88
Anwendung ausführen und debuggen . . . . .	89
Anwendung starten . . . . .	89
Fehler in der Entwicklerkonsole entdecken . . . . .	90
Die Anwendung im Browser debuggen . . . . .	91
Debuggen mit Visual Studio Code . . . . .	92
Zusammenfassung . . . . .	94
<b>4 Komponenten und Datenbindung . . . . .</b>	<b>95</b>
Datenbindung in Angular . . . . .	95
Rückblick auf AngularJS 1.x . . . . .	95
Property-Binding . . . . .	96
Event-Bindings . . . . .	97
Das Zusammenspiel von Property- und Event-Bindings . . . . .	98
Bindings im Template . . . . .	99
Two-Way-Bindings . . . . .	99

Eigene Komponenten mit Datenbindung .....	100
Überblick .....	101
Vorbereitungen .....	101
Eine Komponente mit Property-Bindings .....	103
Komponenten mit Event-Bindings .....	108
Komponenten mit Two-Way-Bindings .....	111
Life-Cycle-Hooks .....	112
Ausgewählte Hooks .....	112
Experiment mit Life-Cycle-Hooks .....	113
Angular und Zyklen .....	115
DateControl mit Life-Cycle-Hooks .....	117
Zusammenfassung .....	121
<b>5 Services und Dependency Injection .....</b>	<b>123</b>
Ein erster Service .....	123
Services austauschen .....	128
Services mit klassischen Providern konfigurieren .....	133
Einen Service lokal registrieren .....	135
Arten von Providern .....	137
useClass .....	138
useValue .....	138
useFactory .....	140
useExisting .....	141
multi .....	142
Konstanten als Tokens .....	145
Zusammenfassung .....	147
<b>6 Pipes .....</b>	<b>149</b>
Überblick .....	149
Built-in-Pipes .....	149
Eigene Pipes .....	150
Pure Pipes .....	151
Implementierung einer einfachen Pipe .....	151
Pipes registrieren und nutzen .....	154
Weiterführende Konstellationen .....	155
Pipes und Objekte .....	155
Pipes und Direktiven .....	157
Pipes und Services .....	158
Aufräumarbeiten mit ngOnDestroy .....	159
Zusammenfassung .....	160

<b>7</b>	<b>Module . . . . .</b>	<b>161</b>
	Motivation . . . . .	161
	Eine Angular-typische Modulstruktur . . . . .	162
	Shared Modules . . . . .	163
	Feature-Modules . . . . .	167
	Root-Modules . . . . .	169
	Module reexportieren . . . . .	170
	Zusammenfassung . . . . .	172
<b>8</b>	<b>Routing . . . . .</b>	<b>173</b>
	Überblick . . . . .	173
	Erste Schritte mit dem Router . . . . .	175
	Routing-Konfiguration für das AppModule einrichten . . . . .	176
	Routing-Konfiguration für Feature-Modules einrichten . . . . .	178
	Platzhalter in AppComponent hinterlegen . . . . .	179
	Hyperlinks zum Aktivieren von Routen einrichten . . . . .	180
	Parametrisierte Routen . . . . .	182
	Arten von Routing-Parametern . . . . .	182
	Parameter in Komponenten auslesen . . . . .	182
	Parametrisierte Routen konfigurieren . . . . .	184
	Auf parametrisierte Routen verweisen . . . . .	185
	Hierarchisches Routing mit Child-Routes . . . . .	186
	Überblick über Child-Routes . . . . .	186
	Child-Komponente implementieren . . . . .	187
	Child-Komponente registrieren . . . . .	189
	Hyperlinks zum Aktivieren von Child-Routen einrichten . . . . .	190
	Aux-Routes . . . . .	191
	Platzhalter für Aux-Routes . . . . .	192
	Komponente für Aux-Route erzeugen . . . . .	193
	Konfiguration für Aux-Route . . . . .	194
	Verweise auf Aux-Routes einrichten . . . . .	195
	Mit dem Query-String und dem Hash-Fragment arbeiten . . . . .	196
	QueryString und Hash-Fragment programmatisch beeinflussen . . . . .	197
	Query-String und Hash-Fragment deklarativ beeinflussen . . . . .	199
	Query-String und Hash-Fragment auslesen . . . . .	200
	HTML5-Routing vs. Hash-Routing . . . . .	201
	PathLocationStrategy . . . . .	201
	HashLocationStrategy . . . . .	203
	Zusammenfassung . . . . .	203

<b>9</b>	<b>Template-getriebene Formulare und Validierung . . . . .</b>	<b>205</b>
	FormsModule einbinden . . . . .	206
	Eingaben validieren . . . . .	206
	Zugriff auf den Zustand des Formulars . . . . .	207
	Bedingte Formatierung von Eingabefeldern . . . . .	212
	Eigene Validierungsdirektiven . . . . .	212
	Eine erste Validierungsdirektive erstellen . . . . .	213
	Parametrisierbare Validierungsdirektiven . . . . .	218
	Multi-Field-Validatoren erstellen . . . . .	221
	Asynchrone Validatoren . . . . .	223
	Komponente zum Präsentieren von Validierungsfehlern . . . . .	226
	Die Standardsteuerelemente von HTML nutzen . . . . .	228
	Checkboxen . . . . .	229
	Radiobuttons . . . . .	229
	Drop-down-Felder . . . . .	230
	Zusammenfassung . . . . .	231
<b>10</b>	<b>Reaktive Formulare . . . . .</b>	<b>233</b>
	Erste Schritte mit reaktiven Formularen . . . . .	233
	Modul einbinden . . . . .	233
	Das Formular mit einem Objektgraphen beschreiben . . . . .	234
	Reactive Formulare mit dem FormBuilder beschreiben . . . . .	236
	Einen Objektgraphen an ein Formular binden . . . . .	237
	Werte ins Formular schreiben . . . . .	239
	Validatoren . . . . .	239
	Synchrone Validatoren . . . . .	240
	Parametrisierte Validatoren . . . . .	242
	Asynchrone Validatoren . . . . .	243
	Multi-Field-Validatoren für reaktive Formulare . . . . .	246
	Geschachtelte Formulare . . . . .	248
	Geschachtelte FormGroup . . . . .	248
	Wiederholgruppen mit FormArray . . . . .	250
	Dynamische Formulare . . . . .	253
	Zusammenfassung . . . . .	254
<b>11</b>	<b>Reactive Extensions Library for JavaScript (RxJS) . . . . .</b>	<b>255</b>
	Grundlegende Typen von RxJS . . . . .	255
	Observables, Observer und Operatoren . . . . .	255
	Observables instanziieren . . . . .	258
	Subjects . . . . .	261

Observables vs. Promises . . . . .	262
Observables in Promises umwandeln . . . . .	263
Promises in Observables umwandeln . . . . .	263
Gruppen von Operatoren . . . . .	264
Creation Operators . . . . .	264
Transformation Operators . . . . .	265
Filtering Operators . . . . .	265
Join Operators . . . . .	266
Error Handling Operators . . . . .	267
Multicasting Operators . . . . .	267
Utility Operators . . . . .	267
Reaktiver Entwurf . . . . .	267
Flattening . . . . .	271
Datenflüsse kombinieren . . . . .	272
Der Operator combineLatest . . . . .	272
combineLatest vs. withLatestFrom . . . . .	275
Der Operator merge . . . . .	277
Multicasting . . . . .	279
Motivation für Multicasting . . . . .	279
Hot vs. Cold Observables . . . . .	281
Fehlerbehandlung . . . . .	282
Observables schließen . . . . .	285
Reaktive Services . . . . .	287
Zusammenfassung . . . . .	291
<b>12 Testautomatisierung . . . . .</b>	<b>293</b>
Jasmine und Karma . . . . .	293
Aufbau eines Jasmine-Tests . . . . .	293
Tests mit Karma ausführen . . . . .	295
Karma auf dem Build-Server . . . . .	296
Angular und Jasmine . . . . .	297
Komponenten mit dem TestBed testen . . . . .	297
Arbeiten mit Attrappen (Mocks) . . . . .	299
Gray-Box-Tests mit Spys . . . . .	304
HTTP-Zugriffe simulieren . . . . .	306
Asynchrone Tests . . . . .	308
Templates mit DOM-Zugriffen testen . . . . .	310
Direktiven testen . . . . .	312
Pipes testen . . . . .	312
Testabdeckung ermitteln . . . . .	313
Zusammenfassung . . . . .	314

<b>13. Performancetuning</b> . . . . .	<b>315</b>
Optimierte Datenbindung mit OnPush . . . . .	315
Datenbindung visualisieren . . . . .	316
Immutables . . . . .	317
Immutables und Datenbindung . . . . .	319
Observables und Datenbindung . . . . .	319
Immutables und/oder Observables . . . . .	324
Manuelle Änderungsverfolgung . . . . .	324
Lazy Loading von Routen . . . . .	325
Routen für das Lazy Loading einrichten . . . . .	325
Lazy Loading im Browser nachvollziehen . . . . .	327
Lazy Loading und Tree-Shakable Provider . . . . .	328
Lazy Loading, klassische Provider und Shared Modules . . . . .	330
Korrekte Nutzung von SharedModules beim Einsatz von Lazy Loading . . . . .	335
Preloading . . . . .	337
Preloading aktivieren . . . . .	337
Eigene Preloading-Strategien entwickeln . . . . .	339
Selektives Preloading mit eigener Preloading-Strategie . . . . .	340
Zusammenfassung . . . . .	341
<b>14. Querschnittsfunktionen</b> . . . . .	<b>343</b>
Guards . . . . .	343
Das Aktivieren von Routen verhindern . . . . .	344
Das Deaktivieren einer Komponente verhindern . . . . .	346
Events . . . . .	349
Resolver . . . . .	351
Vorbereitungen . . . . .	351
Resolver erzeugen und verwenden . . . . .	352
HttpInterceptoren . . . . .	355
Zusammenfassung . . . . .	358
<b>15. Authentifizierung und Autorisierung</b> . . . . .	<b>359</b>
Cookie-basierte Security . . . . .	359
Cookies und XSRF . . . . .	360
Tokenbasierte Security . . . . .	361
OAuth 2 und OpenID Connect . . . . .	361
OAuth 2 . . . . .	361
Benutzer mit OpenID Connect authentifizieren . . . . .	363
JSON Web Token . . . . .	364
OAuth-2- und OIDC-Flows . . . . .	365
OAuth 2 und OIDC mit Angular nutzen . . . . .	366
Zusammenfassung . . . . .	370

<b>16 Internationalisierung . . . . .</b>	<b>371</b>
I18N mit dem Angular-Compiler . . . . .	371
Überblick . . . . .	372
@angular/localize installieren . . . . .	373
Texte markieren . . . . .	373
Strings in der Komponentenklasse markieren . . . . .	374
Texte extrahieren . . . . .	375
Übersetzte Texte in Builds integrieren . . . . .	377
Sprache beim Einsatz von ng serve festlegen . . . . .	379
Übersetzungstexte zur Laufzeit angeben . . . . .	380
Grammatikalische Formen berücksichtigen . . . . .	382
Unterschiedliche Formate unterstützen . . . . .	383
Manuell weitere Formate laden . . . . .	384
ngx-translate . . . . .	385
Überblick . . . . .	385
Bibliothek installieren und konfigurieren . . . . .	386
Sprachdateien bereitstellen . . . . .	388
Texte einbinden . . . . .	388
Texte zur Laufzeit laden . . . . .	389
Sprachwechsel . . . . .	390
Grammatikalische Formen berücksichtigen . . . . .	391
Unterschiedliche Formate nutzen . . . . .	393
Lazy Loading . . . . .	393
Zusammenfassung . . . . .	394
<b>17 Reaktive Zustandsverwaltung mit NGRX (Redux) . . . . .</b>	<b>397</b>
Zustandsverwaltung mit Services . . . . .	397
Das Redux-Muster . . . . .	398
NGRX einrichten . . . . .	400
Building-Blocks implementieren . . . . .	402
State modellieren . . . . .	402
Actions festlegen . . . . .	405
Reducer definieren . . . . .	405
Effect einrichten . . . . .	406
Auf den Store zugreifen . . . . .	407
Debuggen mit dem Store . . . . .	408
Selektoren . . . . .	410
Ein erster Selektor . . . . .	410
Selektoren verschachteln . . . . .	411
Meta-Reducer . . . . .	412
Zusammenfassung . . . . .	413

<b>18 Details zu Komponenten und Direktiven . . . . .</b>	<b>415</b>
Vorbereitungen . . . . .	415
Weiterführende Aspekte von Komponenten . . . . .	418
Content Projection . . . . .	418
Parent-Komponenten referenzieren . . . . .	420
View und Content . . . . .	424
Kommunikation über Template-Variablen . . . . .	434
Kommunikation über Services . . . . .	434
Attributdirektiven . . . . .	438
Direktiven definieren . . . . .	438
Mit der Umwelt kommunizieren . . . . .	440
Direktiven und Template-Variablen . . . . .	442
Strukturelle Direktiven . . . . .	443
Templates und Container . . . . .	443
Microsyntax . . . . .	445
Eine einfache DataTable umsetzen . . . . .	447
ViewContainerRef direkt zum Einblenden von Templates verwenden . . . . .	452
Bestehende ViewContainer ergänzen . . . . .	453
Dialoge dynamisch einblenden . . . . .	454
ViewContainerRef direkt zum dynamischen Erzeugen von Komponenten verwenden . . . . .	462
Komponenten für Formularfelder . . . . .	463
Ausgaben formatieren und Eingaben parsen . . . . .	464
Eigene Formularsteuerelemente . . . . .	467
Zusammenfassung . . . . .	470
<b>19 Wiederverwendbare Bibliotheken und Monorepos . . . . .</b>	<b>471</b>
Monorepo erstellen . . . . .	471
Aufbau von Bibliotheken . . . . .	473
Bibliothek in Monorepo ausprobieren . . . . .	475
npm-Paket bauen und bereitstellen . . . . .	478
npm-Paket konsumieren . . . . .	480
Zusammenfassung . . . . .	480
<b>Index . . . . .</b>	<b>481</b>



---

# Vorwort

Vor etwas mehr als zehn Jahren galt für gute Webanwendungen noch die Regel, so viele Aufgaben wie möglich auf dem Server zu erledigen. Inzwischen stützen sich moderne Webanwendungen jedoch auf clientseitige Techniken, allen voran JavaScript. Dies steigert die Benutzerfreundlichkeit und schafft die Möglichkeit, die jeweilige Anwendung an die Auflösungen und Formfaktoren der vielen unterschiedlichen klassischen und mobilen Plattformen anzupassen.

*Single Page Applications* (SPAs) bilden einen derzeit äußerst beliebten Architekturstil für solche Webanwendungen. Wie ihr Name schon vermuten lässt, bestehen SPAs aus lediglich einer einzigen Seite, die ein Browser auf klassischem Weg abruft und anzeigt. Alle weiteren Seiten und Daten bezieht die SPA bei Bedarf über direkte HTTP-Zugriffe per JavaScript.

Das populäre JavaScript-Framework Angular, das von Google entwickelt wird, hilft Ihnen beim Erstellen von Anwendungen, die diesen Architekturstil verfolgen. Angular bietet unter anderem Unterstützung für die Datenbindung, für das Validieren von Daten sowie das Arbeiten mit Vorlagen. Darüber hinaus stellt Angular Konzepte zur Verfügung, mit denen Sie den Quellcode strukturieren und wartbare, wiederverwendbare sowie testbare Programmteile erschaffen können. Das vorliegende Buch präsentiert die Möglichkeiten von Angular. Dabei beschränkt es sich nicht nur auf die Grundlagen, sondern geht auch auf die zugrunde liegenden Konzepte ein.

## Zielgruppe

Das Buch richtet sich an Entwickler, die bereits grundlegende Erfahrungen mit HTML, CSS und JavaScript gesammelt haben und nun mit Angular SPAs entwickeln wollen. Dabei bezieht es sich auf die Version 12 von Angular, die beim Verfassen der Texte die aktuellste Version war. Aufgrund des evolutionären Charakters von Angular gehen wir jedoch davon aus, dass dieses Buch auch für viele darauf folgende Versionen geeignet ist.

# Zielsetzung des Buchs

Mit diesem Buch verfolgen wir das Ziel, Ihnen anhand von Beispielen zu zeigen, wie Sie Angular zur Entwicklung von Single Page Applications nutzen können. Dabei gehen wir auf die Möglichkeiten von Angular ein und präsentieren auch Lösungen für Aspekte, die Angular nicht direkt unterstützt.

## Quellcodebeispiele, Onlineservices und Errata

Über <http://www.ANGLARarchitects.io/leser> stellen wir Ihnen sämtliche in diesem Buch präsentierte Quellcodebeispiele sowie Web-APIs zur Verfügung, die die Beispiele zu Demonstrationszwecken nutzen und die Sie auch in eigene Projekte einbinden können. Darüber hinaus werde ich auf dieser Seite weitere Informationen zu Angular sowie gegebenenfalls Errata zum vorliegenden Buch veröffentlichen. Daneben bietet die Website Ihnen die Möglichkeit, mit mir als Autor direkt in Kontakt zu kommen.

## Konventionen

### Kursiv

Wird genutzt für neue Begriffe, URLs, Dateinamen und E-Mail-Adressen

### Nichtproportionalschrift

Programmlistings und Codeelemente im Fließtext wie Methoden, Module o.Ä. werden in dieser Schrift dargestellt.



Dieses Symbol steht für Hinweise und allgemeinere Anmerkungen.



Dieses Symbol steht für Tipps.

## Aufbau des Buchs

Das Buch besteht aus 19 Kapiteln. Die folgende Auflistung zeigt, was diese bieten:

- Kapitel 1, *Projekt-Setup*: In diesem Kapitel erfahren Sie, welche Schritte nötig sind, um mit Angular loszulegen. Dazu lernen Sie unter anderem das *Angular Command Line Interface* (CLI) kennen. Damit generieren wir auch schon das Grundgerüst für unsere erste Angular-Anwendung.

- Kapitel 2, *Erste Schritte mit TypeScript*: Die bevorzugte Sprache zum Entwickeln mit Angular ist TypeScript. Sie orientiert sich am ECMAScript-Standard und bietet zusätzlich ein statisches Typsystem, um Fehler frühzeitig erkennen zu können. In diesem Kapitel lernen Sie die wichtigsten Merkmale dieser JavaScript-Erweiterung kennen.
- Kapitel 3, *Eine erste Angular-Anwendung*: Dieses Kapitel zeigt Ihnen, wie Sie mit Angular eine erste einfache Anwendung erstellen können, die via HTTP Daten abruft und darstellt. Darüber hinaus lernen Sie hier die Grundlagen zu Komponenten und Modulen kennen. Auch in das Thema Datenbindung wird eingeführt.
- Kapitel 4, *Komponenten und Datenbindung*: Angular-Anwendungen sind Komponenten, die aus weiteren Komponenten bestehen. In diesem Kapitel erfahren Sie, wie Sie eigene Komponenten erstellen können, die über Datenbindung mit anderen Komponenten kommunizieren.
- Kapitel 5, *Services und Dependency Injection*: Services sind unter Angular wiederverwendbare Codeeinheiten. Dank Dependency Injection können Sie sie austauschbar gestalten. Dieses Kapitel zeigt die Möglichkeiten zum Entwickeln solcher Services auf. Außerdem erfahren Sie hier, wie Sie Services global oder auch nur für bestimmte Komponenten registrieren können.
- Kapitel 6, *Pipes*: Pipes helfen dabei, Daten im Rahmen der Datenbindung zu transformieren. Sie kommen zum Beispiel zum Formatieren oder Umschlüsse von Informationen zum Einsatz. Dieses Kapitel informiert über die in Angular enthaltenen Pipes und erklärt, wie Sie in Ihren Projekten eigene Pipes schreiben können.
- Kapitel 7, *Module*: Zum Strukturieren von Anwendungen setzt Angular auf Module. In diesem Kapitel lernen Sie eine typische Modulstruktur für Angular-Anwendungen kennen. Außerdem erfahren Sie, wie Sie eigene Module umsetzen können.
- Kapitel 8, *Routing*: Mit Routing lassen sich in einer Single Page Application mehrere Seiten simulieren und so Navigationsstrukturen etablieren. Dieses Kapitel geht auf den Angular-Router ein, der sich um diese Aufgabe kümmert.
- Kapitel 9, *Template-getriebene Formulare und Validierung*: In diesem Kapitel lernen Sie, Template-getriebene Formulare mit Angular aufzubauen. Außerdem erfahren Sie, wie Sie Eingaben validieren und eigene Formularsteuerelemente anbieten können.
- Kapitel 10, *Reaktive Formulare*: Reaktive Formulare bilden eine sehr mächtige Alternative zu den im vorangegangenen Kapitel präsentierten Template-getriebenen Formularen. Hier erfahren Sie, was es damit auf sich hat.
- Kapitel 11, *Reactive Extensions Library for JavaScript (RxJS)*: Angular nutzt die von der Bibliothek RxJS angebotenen Observables zur Darstellung asynchroner Operationen. In diesem Kapitel werfen wir einen etwas genaueren Blick auf

*RxJS* und die zugrunde liegenden Konzepte. Wir machen uns mit den verschiedenen Gruppen von Operatoren vertraut und lernen populäre Vertreter dieser Gruppen kennen. Außerdem beschäftigen wir uns mit den Schritten des reaktiven Entwurfs, mit Hot und Cold Observables bzw. Multicasting sowie mit Fehlerbehandlung, Flattening und dem Kombinieren verschiedener Datenflüsse.

- Kapitel 12, *Testautomatisierung*: Wie die Qualität von Angular-Code mit automatisierten Tests geprüft werden kann, erfahren Sie in diesem Kapitel. Wir gehen dazu auf das populäre Testing-Framework Jasmine in Kombination mit den Angular Testing Utilities ein und zeigen Ihnen, wie Sie damit Tests für Ihren Angular-Code schreiben können. Begriffe und Techniken wie Mocks und Spys werden ebenfalls thematisiert.
- Kapitel 13, *Performancetuning*: In diesem Kapitel betrachten wir zwei elementare Stellschrauben, die die Performance von Angular-Anwendungen beeinflussen, und lernen dabei auch die Art und Weise, wie das Framework arbeitet, besser kennen. Zum einen betrachten wir die Datenbindungsstrategie OnPush und nutzen sie gemeinsam mit Observables und Immutables, zum anderen beschäftigen wir uns mit Lazy-Loading- und Preloading-Strategien.
- Kapitel 14, *Querschnittsfunktionen*: Querschnittsfunktionen sind meist technische Anforderungen, die es immer und immer wieder zu berücksichtigen gilt. Beispiele dafür sind unter anderem Authentifizierung, Protokollierung und die Behandlung von Fehlern. In diesem Kapitel beschäftigen wir uns mit Mechanismen zur Implementierung solcher Querschnittsfunktionen, unter anderem mit Guards, Resolver und HttpInterceptoren.
- Kapitel 15, *Authentifizierung und Autorisierung*: Die wenigsten Anwendungen kommen ohne Authentifizierung und Autorisierung aus. Bei Single Page Applications bieten sich hierzu neben Cookies auch tokenbasierte Verfahren an. Dieses Kapitel beschreibt beide Optionen. Für Letztere geht es auf die populären Standards OAuth 2 und OpenID Connect ein und zeigt, wie Sie damit ein Security-Token anfordern können. Anschließend erfahren Sie, wie sich dieses Vorgehen in Ihren Angular-Anwendungen umsetzen lässt.
- Kapitel 16, *Internationalisierung*: In diesem Kapitel zeigen wir Ihnen, wie das Anpassen von Angular-Anwendungen für Benutzer verschiedener Länder und Sprachen funktioniert. Die dafür in den Angular-Compiler integrierten Möglichkeiten betrachten wir dabei genauso wie den Einsatz der populären Bibliothek *ngx-translate*.
- Kapitel 17, *Reaktive Zustandsverwaltung mit NGRX (Redux)*: State Management hilft beim Verwalten lokal vorgehaltener Daten, auf die unter anderem mehrere Komponenten zugreifen. Die wohl populärste Bibliothek, die dabei unterstützt, ist *NGRX*. In diesem Kapitel erfahren Sie, was sich dahinter verbirgt und wie Sie diese Bibliothek in Ihrer Angular-Anwendung nutzen können.

- Kapitel 18, *Details zu Komponenten und Direktiven*: Obwohl seit der Angular-Einführung in Kapitel 3 ständig Komponenten zum Einsatz kommen, gibt es doch noch einige Details, die unter anderem bei der Entwicklung wiederverwendbarer Komponenten nützlich sind. Diese lernen Sie hier kennen. Außerdem erfahren Sie, wie sich mit Direktiven wiederkehrende Aufgaben umsetzen lassen.
- Kapitel 19, *Wiederverwendbare Bibliotheken und Monorepos*: In diesem Kapitel erfahren Sie, wie sich Ihre Angular-Anwendungen mittels Bibliotheken in einem mit der Angular CLI generierten Monorepo untergliedern lassen, aber auch, wie sie diese Bibliotheken als wiederverwendbare npm-Pakete für andere Projektteams veröffentlichen können.

## Schulungen und Beratung

Der Autor bietet Schulungen und Beratung zu Angular an. Informationen dazu finden Sie unter <http://www.ANGLULARarchitects.io>.

## Danksagungen

Meinen Dank für ihre Mitwirkung an diesem Buch möchte ich aussprechen an

- Ariane Hesse, Lektorat, und Sibylle Feldmann, Korrektorat, die das Manuskript bearbeitet haben.
- Hans-Peter Grahsl, der sehr sorgfältig und unter vollstem Einsatz seines berüchtigten Adlerblicks äußerst wertvolles inhaltliches sowie fachliches Feedback zu den Texten und den Beispielen gegeben hat.



# KAPITEL 1

# Projekt-Setup

Moderne JavaScript-Projekte gleichen immer mehr klassischen Anwendungen: Sie nutzen Compiler, um moderne typsichere Sprachen wie TypeScript in handelsübliches JavaScript zu übersetzen. Zusätzlich verwenden sie Werkzeuge, mit denen sie optimierte Bundles erzeugen. Damit sind JavaScript-Dateien gemeint, die sich aus mehreren einzelnen Dateien zusammensetzen.

Durch dieses Vorgehen müssen nur noch wenige Dateien auf dem Server platziert sowie in den Browser geladen werden. Ersteres erhöht den Komfort beim Deployment, und Letzteres verbessert die Startgeschwindigkeit der Anwendung. Außerdem kommen in modernen JavaScript-Projekten auch Werkzeuge zur Testautomatisierung zum Einsatz.

Dieses Kapitel zeigt, wie sich ein Projekt-Setup für Angular, das diesen Kriterien genügt, einrichten lässt. Es beginnt mit der Installation und Einrichtung von Visual Studio Code, der in diesem Buch verwendeten Entwicklungsumgebung. Danach wendet sich das Kapitel dem *Angular Command Line Interface* (CLI) zu. Dabei handelt es sich um eine vom Angular-Team bereitgestellte Konsolenanwendung, die viele Aufgaben rund um die Angular-Entwicklung automatisiert. Schließlich erfahren Sie in diesem Kapitel auch, wie ein mit der CLI generiertes Projekt aufgebaut ist.

## Visual Studio Code

Wir nutzen in diesem Buch die freie Entwicklungsumgebung Visual Studio Code (<https://code.visualstudio.com>). Sie funktioniert auf allen wichtigen Betriebssystemen (Linux, macOS, Windows) und ist äußerst leichtgewichtig. Visual Studio Code unterstützt auch die Sprache TypeScript. Bei dieser handelt es sich um eine typsichere Obermenge von JavaScript, die für die Angular-Entwicklung verwendet wird.

Außerdem existieren zahlreiche Erweiterungen, die die Arbeit mit Frameworks wie Angular vereinfachen. Um Erweiterungen zu installieren, klicken Sie auf das Symbol *Extensions* in der linken Symbolleiste. Anschließend können Sie nach Erweiterungen suchen und diese installieren (siehe Abbildung 1-1).

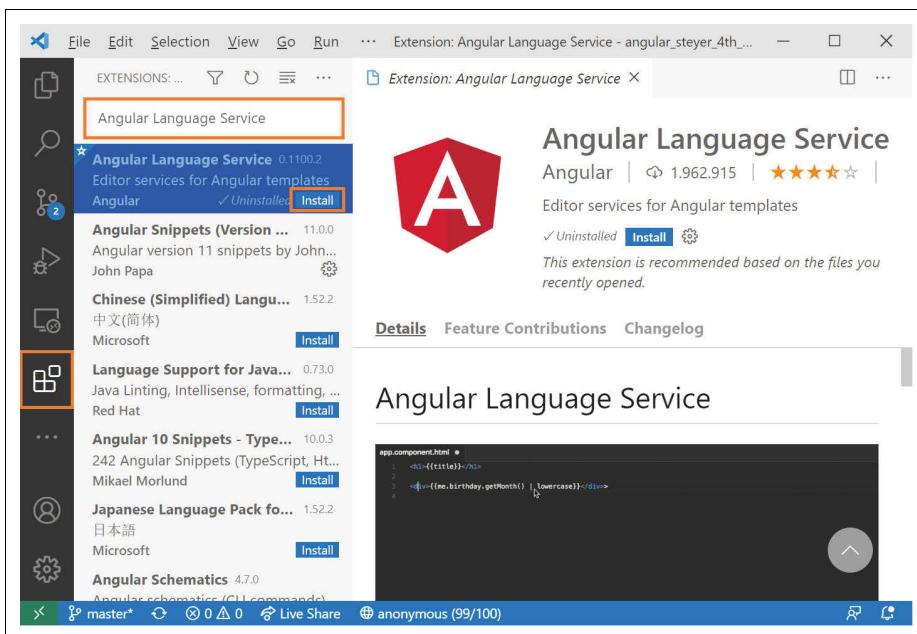


Abbildung 1-1: Erweiterungen in Visual Studio Code installieren

Für die Entwicklung von Angular-Lösungen empfehlen wir die folgenden Erweiterungen:

#### *Angular Language Service*

Der Angular Language Service wird vom Angular-Team bereitgestellt und erlaubt Angular-bezogene Codevervollständigungen in HTML-Templates. Außerdem weist der Language Service auch auf mögliche Fehler in HTML-Templates hin.

#### *Angular Schematics*

Erlaubt das Generieren von Building-Blocks wie Angular-Komponenten über das Kontextmenü von Visual Studio Code.

#### *Debugger for Chrome*

Erlaubt das Debuggen von JavaScript-Anwendungen, die in Chrome ausgeführt werden.

Bitte installieren Sie diese Erweiterungen. Wir werden bei Bedarf in den einzelnen Kapiteln darauf zurückkommen.



Neben Visual Studio Code haben wir auch mit den kommerziellen Produkten WebStorm, PhpStorm bzw. IntelliJ von JetBrains (<https://www.jetbrains.com/>) sehr gute Erfahrungen gemacht. Es handelt sich bei diesen drei Lösungen eigentlich um das gleiche Produkt in verschiedenen Ausprägungen. PhpStorm unterstützt zum Beispiel darüber hinaus PHP, während IntelliJ zusätzlich viele Annehmlichkei-

ten für Java-Entwickler mit sich bringt. Diese Lösungen sind zwar etwas schwergewichtiger als Visual Studio Code, bieten dafür jedoch ab Werk zahlreiche Features, wie umfangreiche Refactoring-Möglichkeiten oder Test-Runner für Unit-Tests.

Tatsächlich sind Visual Studio Code und WebStorm/IntelliJ mit Abstand die am häufigsten eingesetzten Entwicklungsumgebungen, auf die wir bei unseren Kundenprojekten im Angular-Umfeld stoßen.

## Angular CLI

Um keine Zeit mit dem Einrichten aller benötigten Werkzeuge zu verlieren, bietet das Angular-Team das sogenannte *Angular Commandline Interface*, kurz Angular CLI (<https://cli.angular.io>), an. Die CLI generiert nicht nur das Grundgerüst der Anwendung, sondern auf Wunsch auch die Grundgerüste weiterer Anwendungsbestandteile wie z. B. Komponenten.

Außerdem kümmert sie sich um das Konfigurieren des TypeScript-Compilers und einer Build-Konfiguration zur Erzeugung optimierter Bundles. Werkzeuge für die Testautomatisierung richtet die CLI ebenfalls ein.

## Node.js und Angular CLI installieren

Die CLI lässt sich leicht über den Package-Manager `npm` beziehen, der sich im Lieferumfang von Node.js ([nodejs.org](https://nodejs.org)) befindet. Außerdem nutzt sie Node.js als Laufzeitumgebung. Deswegen sollten Sie zur Vorbereitung eine aktuelle Node.js-Version von Node.js (<https://nodejs.org>) herunterladen und installieren. Die Autoren haben gute Erfahrungen mit den jeweiligen Long-Term-Support-Versionen (LTS-Versionen) gemacht. Der Einsatz älterer Versionen kann zu Problemen führen.

Sobald Node.js installiert ist, kann die CLI mittels `npm` eingerichtet werden:

```
npm install -g @angular/cli
```

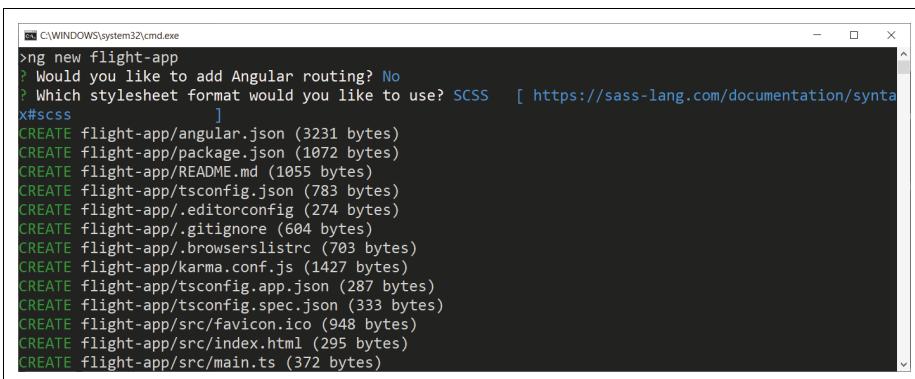
Der Schalter `-g` bewirkt, dass `npm` das Werkzeug systemweit, also global, einrichtet, sodass es überall zur Verfügung steht. Ohne diesen Schalter würde `npm` das adressierte Paket lediglich für ein lokales Projekt im aktuellen Ordner einrichten. Nach der Installation steht die CLI über das Kommando `ng` zur Verfügung.

## Ein Projekt mit der CLI generieren

Ein Aufruf von

```
ng new flight-app
```

generiert das Grundgerüst einer neuen Angular-Anwendung, die den Namen `flight-app` erhält. Dazu stellt es uns ein paar Fragen (siehe Abbildung 1-2):



```
C:\WINDOWS\system32\cmd.exe
>ng new flight-app
? Would you like to add Angular routing? No
? Which stylesheet format would you like to use? SCSS [ https://sass-lang.com/documentation/syntax ]
x#scss
CREATE flight-app/angular.json (3231 bytes)
CREATE flight-app/package.json (1072 bytes)
CREATE flight-app/README.md (1055 bytes)
CREATE flight-app/tsconfig.json (783 bytes)
CREATE flight-app/.editorconfig (274 bytes)
CREATE flight-app/.gitignore (604 bytes)
CREATE flight-app/.browserslistrc (703 bytes)
CREATE flight-app/karma.conf.js (1427 bytes)
CREATE flight-app/tsconfig.app.json (287 bytes)
CREATE flight-app/tsconfig.spec.json (333 bytes)
CREATE flight-app/src/favicon.ico (948 bytes)
CREATE flight-app/src/index.html (295 bytes)
CREATE flight-app/src/main.ts (372 bytes)
```

Abbildung 1-2: `ng new` stellt ein paar Fragen, bevor es ein neues Projekt generiert.

Je nach Angular-Version können diese Fragestellungen etwas variieren. Wir gehen hier von folgenden Einstellungen aus:

#### Add Angular Routing

Diese Frage beantworten wir hier mit `No`. Um das Thema Routing kümmert sich Kapitel 8.

#### Stylesheet Format

Wir empfehlen hier SCSS, eine Obermenge von CSS. Die Angular CLI kompiliert diese Dateien für den Browser nach CSS.

Da `ng new` auch zahlreiche Pakete via `npm` bezieht, kann der Aufruf etwas länger dauern.



Seit Version 12 verwendet die CLI standardmäßig den sogenannten Strict Mode. In diesem Modus führen sowohl der TypeScript-Compiler als auch Angular selbst strengere Code-Prüfungen durch. Hierdurch sollen Programmierfehler rascher entdeckt werden.

Falls Sie diese strengeren Prüfungen nicht verwenden wollen, verwenden Sie den Schalter `--strict`:

```
ng new flight-app --strict false
```

Wir gehen in in diesem Buch jedoch davon, dass der Strict Mode aktiviert ist. Das entspricht auch den Empfehlungen des Angular-Teams.

## Angular-Anwendung starten

Um Ihre Anwendung zu starten, wechseln Sie in den generierten Projektordner. Dort bauen Sie mit `ng serve` die Anwendung und stellen sie über einen Demowebserver bereit:

```
cd flight-app
ng serve -o
```

Der Schalter `-o` öffnet einen Browser, der die Anwendung anzeigt. Standardmäßig findet sich diese Anwendung unter `http://localhost:4200`. Ist Port 4200 schon belegt, erkundigt sich `ng serve` nach einer Alternative. Außerdem nimmt der Schalter `--port` den gewünschten Port gleich beim Start von `ng serve` entgegen:

```
ng serve -o --port 4242
```

Die im Browser angezeigte Anwendung sieht wie in Abbildung 1-3 aus. Auch hier kann es von Version zu Version zu Abweichungen kommen.

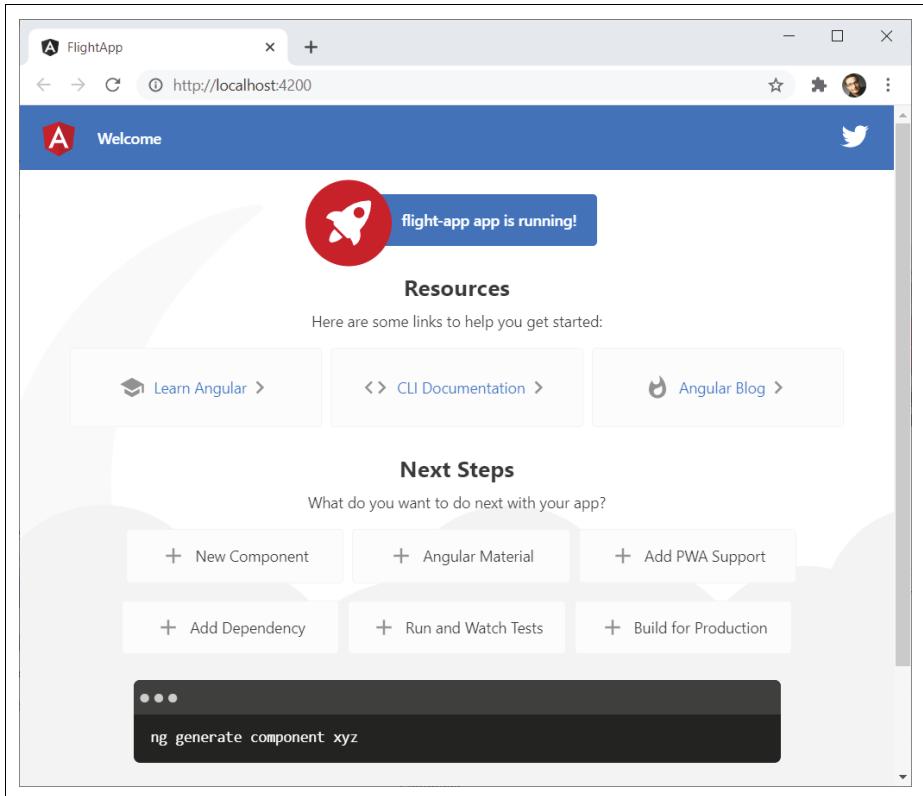


Abbildung 1-3: Generierte Angular-Anwendung

Der für die Entwicklung gedachte Befehl `ng serve` macht aber noch ein wenig mehr: Er überwacht sämtliche Quellcodedateien und stößt das Kompilieren sowie Generieren der Bundles erneut an, wenn sie sich ändern. Danach aktualisiert er auch das Browserfenster.

Um das auszuprobieren, können Sie mit Visual Studio Code die Datei `src\app\app.component.html` öffnen und das erste Vorkommen von `Welcome` durch `Hello World!` ändern. Nach dem Speichern der Datei sollte `ng serve` den betroffenen Teil der Anwendung neu kompilieren, bundeln und den Browser aktualisieren (siehe Abbildung 1-4).

The screenshot shows the Visual Studio Code interface. On the left, there's a tree view with icons for files, folders, and other project components. The main editor area displays the code for `app.component.html`. The code includes HTML and an embedded SVG for the Twitter logo. On the right, a terminal window runs the command `ng serve -o`. The output shows the Angular CLI generating a browser application bundle, listing initial chunk files like vendor.js, polyfills.js, styles.css, and main.js, along with their sizes. It also shows the build time and a success message: "Compiled successfully." and "Browser application bundle generation complete."

Abbildung 1-4: Generierte Angular-Anwendung ändern



Wenn Sie Visual Studio Code verwenden, sollten Sie zunächst den Hauptordner Ihrer Angular-Anwendung mit dem Befehl `File/Open Folder` öffnen. Der Hauptordner ist der, der auch die Datei `package.json` beinhaltet. Das stellt sicher, dass Visual Studio Code sämtliche Konfigurationsdateien findet und keine unnötigen Fehler anzeigt.

Danach können Sie die gewünschte Datei über den links angezeigten Explorer suchen und öffnen. Alternativ dazu bietet sich die Tastenkombination `Strg+P` an. Sie öffnet ein kleines Fenster, mit dem man nach der gewünschten Datei suchen kann.

Mit `Strg+Umschalt+C` können Sie übrigens jederzeit eine externe Konsole im aktuellen Ordner öffnen, um z.B. die Angular CLI auszuführen. Die Tastenkombination `Strg+Umschalt+Ö` öffnet hingegen die Konsole als Terminal direkt in Visual Studio Code.



Die automatische Generierung der Bundles nach einer Änderung am Programmcode funktioniert meist ganz gut, aber ab und an kommt die CLI aus dem Tritt. Das ist unter anderem dann der Fall, wenn Sie mehrere Dateien rasch hintereinander speichern. Auch das Umbenennen von Dateien bringt diesen Mechanismus aus dem Konzept.

Abhilfe schafft hier ein erneutes Speichern der betroffenen Dateien oder – wenn alle Stricke reißen – ein Neustart von `ng serve`.

## Build mit CLI

Während `ng serve` für die Entwicklung sehr komfortabel ist, eignet es sich nicht für den Produktiveinsatz. Um Bundles für die Produktion zu generieren, nutzen Sie die Anweisung

```
ng build
```

Seit Angular CLI 12 führt `ng build` zahlreiche Optimierungen, die zu kleineren Bundles führen, automatisch durch. Davor musste man diese Optimierungen explizit mit dem Schalter `--prod` anfordern.

Ein Beispiel für eine solche Optimierung ist die Minifizierung, bei der unnötige Zeichen wie Kommentare oder Zeilenschaltungen entfernt sowie Ihre Anweisungen durch kompaktere Darstellungsformen ersetzt werden. Ein weiteres Beispiel ist das sogenannte Tree-Shaking, das nicht benötigte Framework-Bestandteile identifiziert und entfernt. Diese Optimierungen verlangsamen natürlich den Build-Prozess ein wenig.

Die generierten Bundles finden sich im Ordner `dist/light-app`. Im Rahmen der Bereitstellung müssen Sie diese Dateien lediglich auf den Webserver Ihrer Wahl kopieren. Da es sich aus Sicht des Webservers hierbei um eine statische Webanwendung handelt, müssen Sie dort auch keine zusätzliche Skriptsprache und kein Web-Framework installieren.

## Projektstruktur von CLI-Projekten

Die von der CLI generierte Projektstruktur orientiert sich an Best Practices, die sich auch in anderen Projekten finden. Für einen ersten Überblick präsentiert Tabelle 1-1 die wichtigsten Dateien. Wir gehen im Laufe des Buchs auf diese und weitere Dateien bei Bedarf genauer ein.

Tabelle 1-1: Projektstruktur

Ordner/Datei	Beschreibung
<code>src/</code>	Beinhaltet alle Quellcodedateien (TypeScript, HTML, CSS etc.).
<code>src/main.ts</code>	Dieser Quellcodedatei kommt besondere Bedeutung zu. Die CLI nutzt sie als Einstiegspunkt in die Anwendung. Deswegen wird ihr Code beim Programmstart zuerst ausgeführt. Standardmäßig beinhaltet sie ein paar Zeilen zum Starten von Angular. Normalerweise müssen Sie diese Datei nicht anpassen.
<code>src/styles.scss</code>	Hier können Sie Ihre eigenen globalen Styles eintragen. Die Dateiendung, z. B. <code>.css</code> oder <code>.scss</code> , hängt von der beim Generieren des Projekts gewählten Option ab.
<code>src/app/</code>	Dieser Ordner und seine Unterordner beinhalten die entwickelten Programmdateien wie zum Beispiel Angular-Komponenten.
<code>src/assets/</code>	Ordner mit statischen Dateien, die die CLI beim Build in das Ausgabeverzeichnis kopiert. Hier könnten Sie zum Beispiel Bilder oder JSON-Dateien ablegen.
<code>dist/</code>	Beinhaltet die von <code>ng build</code> generierten Bundles für die Auslieferung auf einen Server. Der Einsatz von <code>ng serve</code> schreibt diese Bundles hingegen nicht auf die Platte, sondern hält sie lediglich im Hauptspeicher vor.
<code>node_modules/</code>	Beinhaltet sämtliche Module, die über <code>npm</code> bezogen wurden. Dazu gehören der TypeScript-Compiler und andere Werkzeuge für den Build, aber auch sämtliche Bibliotheken für Angular.
<code>tsconfig.json</code>	Konfigurationsdatei für TypeScript. Hier wird zum Beispiel festgelegt, dass der TypeScript-Compiler Ihren Quellcode nach ECMAScript 2015 kompilieren soll. Das ist jene JavaScript-Version, die von allen modernen Browsern der letzten Jahre unterstützt wird.

Tabelle 1-1: Projektstruktur (Fortsetzung)

Ordner/Datei	Beschreibung
<code>.browserslistrc</code>	Listet sämtliche Browser, die die Angular-Anwendung unterstützen soll. Aus dieser Liste ermittelt die Angular CLI nötige CSS-Präfixe, die es beim Kompilieren einfügt. Bis Angular 12 wurde aus dieser Datei auch abgeleitet, ob zusätzlich Legacy-Bundles (ECMAScript 5) für Browser wie Internet Explorer 11 zu erzeugen sind.
<code>package.json</code>	Referenziert sämtliche Bibliotheken, die benötigt werden, inklusive der gewünschten Versionen. Wenn Sie einen Blick auf die Abschnitte <code>dependencies</code> und <code>devDependencies</code> werfen, sehen Sie alle von <code>ng new</code> installierten Pakete. Da der Ordner <code>node_modules</code> mit diesen Paketen nicht in die Quellcodeverwaltung eingecheckt wird, sind diese Hinweise notwendig. Ihre Entwickler-Kolleginnen und -Kollegen müssen lediglich die Anweisung <code>npm install</code> ausführen, um sie in den <code>node_modules</code> -Ordner zu laden.
<code>index.html</code>	Die Startseite. Der Build-Prozess erweitert sie um Referenzen auf die generierten Bundles.
<code>angular.json</code>	Mit dieser Datei lässt sich das Verhalten der CLI anpassen. Beispielsweise referenziert sie globale Styles oder Skripte, die es einzubinden gilt.

Lassen Sie uns nun ein paar der Programmdateien unter `src/app` etwas genauer betrachten. Starten wir dabei mit der generierten `AppComponent`. Wie die meisten Angular-Komponenten besteht sie aus mehreren Dateien:

#### `app.component.ts`

TypeScript-Datei, die das Verhalten der Komponente definiert.

#### `app.component.html`

HTML-Datei mit der Struktur der Komponente.

#### `app.component.scss`

Datei mit lokalen Styles für die Komponente. Allgemeine Styles können in die besprochene `styles.scss` eingetragen werden.

Beispiel 1-1 zeigt den Inhalt der generierten `app.component.ts`:

Beispiel 1-1: Die generierte `app.component.ts`

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {
  title = 'flight-app';
}
```

Es handelt sich dabei um eine Klasse, die lediglich eine Eigenschaft `title` vom Typ `string` besitzt. Letzteres muss hier gar nicht explizit angegeben werden: TypeScript kann sich diesen Umstand aus dem zugewiesenen Standardwert herleiten.

Die Angabe von `export` definiert, dass die Klasse auch in anderen Dateien der Anwendung genutzt werden darf.

Die Klasse wurde mit dem Dekorator `Component` versehen. Dekoratoren definieren Metadaten für Programmkonstrukte wie z. B. Klassen. Diesen importiert die Komponente in der ersten Zeile aus dem Paket `@angular/core`. Bei der Nutzung eines Dekorators wird ihm das Symbol `@` vorangestellt.

Die Metadaten beinhalten den Selektor der Komponente. Das ist in der Regel der Name eines HTML-Elements, das die Komponente repräsentiert. Um die Komponente aufzurufen, können Sie also die folgende Schreibweise in einer HTML-Datei verwenden:

```
<app-root></app-root>
```

Der Dekorator verweist außerdem auf das HTML-Template der Komponente und ihre SCSS-Datei mit lokalen Styles. Letztere ist standardmäßig leer. Die HTML-Datei beinhaltet den Code für die oben betrachtete Startseite. Die ist zwar schön, enthält aber eine Menge HTML-Markup. Ersetzen Sie mal zum Ausprobieren den *gesamten* Inhalt dieser HTML-Datei durch folgendes Fragment:

```
<h1>{{title}}</h1>
```

Wenn Sie nun die Anwendung starten (`ng serve -o`), sollten Sie den Inhalt der Eigenschaft `title` als Überschrift sehen. Die beiden geschweiften Klammernpaare definieren eine sogenannte Datenbindung. Angular bindet also die angegebene Eigenschaft an die jeweilige Stelle im Template.

Mehr Informationen zu TypeScript, Datenbindungen und Angular im Allgemeinen finden Sie in den nächsten beiden Kapiteln. Um diesen Rundgang durch die generierten Programmdateien abzuschließen, möchten wir jedoch noch auf drei weitere generierte Dateien hinweisen. Eine davon ist die Datei `app.module.ts`, die ein Angular-Modul beinhaltet (siehe Beispiel 1-2).

#### *Beispiel 1-2: Das generierte AppModule*

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

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

Angular-Module sind Datenstrukturen, die zusammengehörige Building-Blocks wie Komponenten zusammenfassen. Technisch gesehen, handelt es sich dabei um eine

weitere Klasse. Sie ist in den meisten Fällen leer und dient lediglich als Träger von Metadaten, die über den `NgModule`-Dekorator angegeben werden.

Lassen Sie uns einen Blick auf die Eigenschaften von `NgModule` werfen:

#### `declarations`

Definiert die Inhalte des Moduls. Derzeit beschränken sich diese auf unsere `AppComponent`. Sie wird in der dritten Zeile unter Angabe eines relativen Pfads, der auf die Datei `app.component.ts` verweist, importiert. Die Dateiendung `.ts` wird hierbei weggelassen.

#### `imports`

Importiert weitere Module. Das gezeigte Beispiel importiert lediglich das `BrowserModule`, das alles beinhaltet, um Angular im Browser auszuführen. Das ist auch der Standardfall.

#### `providers`

Hier könnte man sogenannte Services, die Logiken für mehrere Komponenten anbieten, registrieren. Kapitel 5 geht im Detail darauf ein.

#### `bootstrap`

Diese Eigenschaft verweist auf sämtliche Komponenten, die beim Start der Anwendung zu erzeugen sind. Häufig handelt es sich dabei lediglich um eine einzige Komponente. Diese sogenannte Root-Component repräsentiert die gesamte Anwendung und ruft dazu weitere Komponenten auf.

Das Modul, das die Root-Component bereitstellt, wird auch als Root-Module bezeichnet. Angular nimmt es beim Start der Anwendung entgegen und rendert die darin zu findende Root-Component. Für diese Aufgabe hat die CLI die Datei `main.ts` eingerichtet (siehe Beispiel 1-3).

#### *Beispiel 1-3: Die generierte Datei main.ts*

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));
```

Die Funktion `platformBrowserDynamic` erzeugt eine sogenannte Plattform, die die Ausführung von Angular im Browser möglich macht. Andere Plattformen ermöglichen zum Beispiel die serverseitige Ausführung von Angular oder die Ausführung in mobilen Anwendungen. Die Nutzung im Browser ist jedoch der hier betrachtete Standardfall.

Die Methode `bootstrapModule` nimmt das Root-Modul entgegen, und Angular rendert daraufhin ihre Root-Component.

Die verwendete Eigenschaft `environment.production` informiert darüber, ob mit `ng build` ein Build für die Produktion angefordert wird oder mit `ng serve` eines fürs Debuggen. Wie oben erwähnt, veranlasst das die CLI, Optimierungen durchzuführen. Wir können aber auch innerhalb der Anwendung darauf reagieren: Hier wird dann zum Beispiel der Produktivmodus von Angular ebenfalls aktiviert. In diesem Modus ist Angular schneller, erzeugt aber auch weniger sowie weniger gut lesbare Fehlermeldungen.

Um festzulegen, wo auf der Seite unsere Root-Component darzustellen ist, ruft die ebenfalls generierte `index.html` sie auf:

```
<body>
  <app-root></app-root>
</body>
```

Beim Build ergänzt die CLI diese `index.html` auch um Verweise auf die erzeugten Bundles. Eines davon beinhaltet den Code der Datei `main.ts`, die die Angular-Anwendung startet.

## Internet Explorer 11

Bis Version 12 hat Angular Internet Explorer 11 unterstützt. Mit Version 12 wurde diese Unterstützung als veraltet (*deprecated*) gekennzeichnet, und mit Version 13 wird die Internet-Explorer-11-Unterstützung entfernt. Dieser Schritt hat sich angeboten, da selbst Microsoft die Unterstützung von Internet Explorer 11 in Produkten wie Office 365 oder SharePoint abkündigt hat. Das Angular-Team kann somit ab Version 13 moderne Webstandards für die Weiterentwicklung aufgreifen.

Selbst wenn Sie mit Angular 12 oder einer vorherigen Version arbeiten, müssen Sie die Unterstützung für Internet Explorer 11 explizit aktivieren. Dazu tragen Sie in der Datei `.browserslistrc` die folgende Zeile ein:

IE 11

Um eine Ausführung der Angular-Anwendung auch im Internet Explorer 11 zu ermöglichen, generiert die CLI neben den modernen Bundles Legacy-Bundles (ECMAScript 5). Deswegen verlangsamt diese Einstellung die Ausführung von `ng build`. Die Anweisung `ng serve` beschränkt sich standardmäßig hingegen auf moderne Bundles und unterstützt somit Internet Explorer 11 nicht. Um das zu ändern, können Sie in der Datei `tsconfig.json` das in der Eigenschaft `target` hinterlegte Kompilierungsziel auf ES5 ändern. Nun erhalten jedoch auch moderne Browser Legacy-Bundles. Leider sind Legacy-Bundles um einiges größer, und das geht zulasten der Startgeschwindigkeit.

Wollen Sie den Internet Explorer 11 hingegen explizit nicht unterstützen, verwenden Sie die folgende Zeile in Ihrer *browserslistrc*:

```
not IE 11
```

## Eine Style-Bibliothek installieren

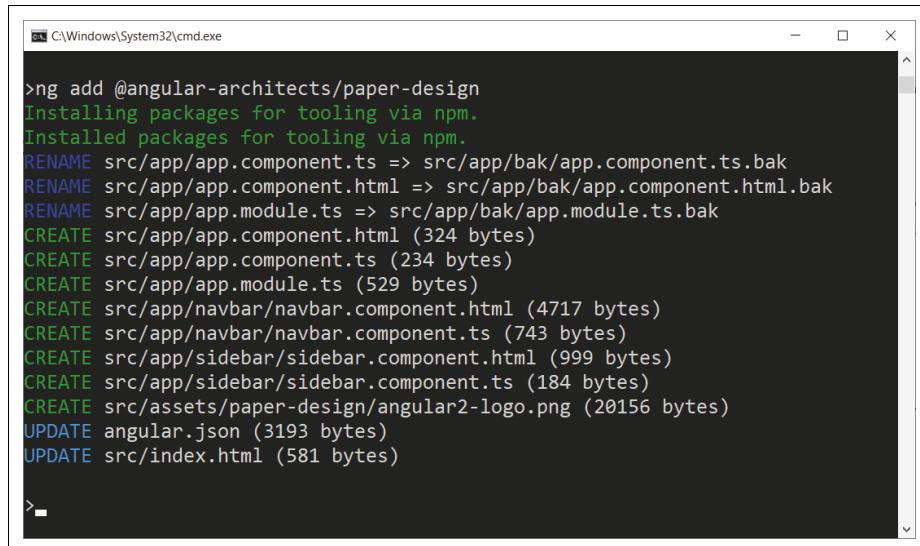
Da auch »das Auge mitprogrammiert«, wollen wir an dieser Stelle ein paar vordefinierte Styles ins Spiel bringen. Es handelt sich dabei um das *Paper-Design-Theme* von Creative Tim (<https://www.creative-tim.com>), das wiederum auf der populären Bibliothek *Bootstrap* (<http://getbootstrap.com>) basiert.

Der Vorteil von Bootstrap liegt neben seiner äußerst weiten Verbreitung in der Tatsache, dass es unaufdringlich ist. Es definiert lediglich ein paar (S)CSS-Klassen, die man auf bekannte HTML-Elemente anwenden kann. Im Gegensatz zu anderen Lösungen muss man also zunächst keine weiteren HTML-Elemente erlernen.

Sie können sowohl *Bootstrap* als auch das *Paper-Design-Theming* mit dem folgenden Befehl installieren:

```
ng add @angular-architects/paper-design
```

Dieses von uns bereitgestellte Paket beinhaltet die besprochenen Styles und generiert auch einige Konfigurationseinträge sowie das Skelett unserer Anwendung (siehe Abbildung 1-5).



```
C:\Windows\System32\cmd.exe
>ng add @angular-architects/paper-design
Installing packages for tooling via npm.
Installed packages for tooling via npm.
RENAME src/app/app.component.ts => src/app/bak/app.component.ts.bak
RENAME src/app/app.component.html => src/app/bak/app.component.html.bak
RENAME src/app/app.module.ts => src/app/bak/app.module.ts.bak
CREATE src/app/app.component.html (324 bytes)
CREATE src/app/app.component.ts (234 bytes)
CREATE src/app/app.module.ts (529 bytes)
CREATE src/app/navbar/navbar.component.html (4717 bytes)
CREATE src/app/navbar/navbar.component.ts (743 bytes)
CREATE src/app/sidebar/sidebar.component.html (999 bytes)
CREATE src/app/sidebar/sidebar.component.ts (184 bytes)
CREATE src/assets/paper-design/angular2-logo.png (20156 bytes)
UPDATE angular.json (3193 bytes)
UPDATE src/index.html (581 bytes)

>-
```

Abbildung 1-5: Generierte Angular-Anwendung

Wie Sie hier sehen, verschiebt dieser Befehl die *AppComponent* und das *AppModule* in den Ordner *bak* (siehe Zeilen mit *RENAME*). Danach generiert er die beiden erneut im Ordner *src/app*. Außerdem generiert er eine *NavBarComponent*, eine *SideBarComponent*

und ein Angular-Logo. Danach erweitert dieser Aufruf von `ng add` die Dateien `angular.json` und `index.html`. Erstere erhält Verweise auf die Style-Dateien (siehe Beispiel 1-4) von *Bootstrap* und dem freien *Paper Design-Theming* von Creative Tim. Letztere erhält zwei `link`-Elemente zum Laden des vom Theming verwendeten Webfonts.

*Beispiel 1-4: Referenzierte Styles in angular.json*

```
"styles": [  
    "node_modules/@angular-architects/paper-design/assets/css/bootstrap.css",  
    "node_modules/@angular-architects/paper-design/assets/scss/paper-dashboard.scss",  
    "src/styles.scss"  
,
```

In Beispiel 1-4 sieht man übrigens auch die von `ng new` generierte Datei `src/styles.scss`, in der Sie Ihre eigenen globalen Styles hinterlegen können.



Leider liest `ng serve` globale Konfigurationsdateien wie die `angular.json` nur beim Programmstart. Falls `-ng serve-` bereits läuft, müssen Sie es deswegen beenden (`Strg+C`) und neu starten.

Startet man die Anwendung erneut mit `ng serve -o`, ergibt sich das in Abbildung 1-6 gezeigte Bild.

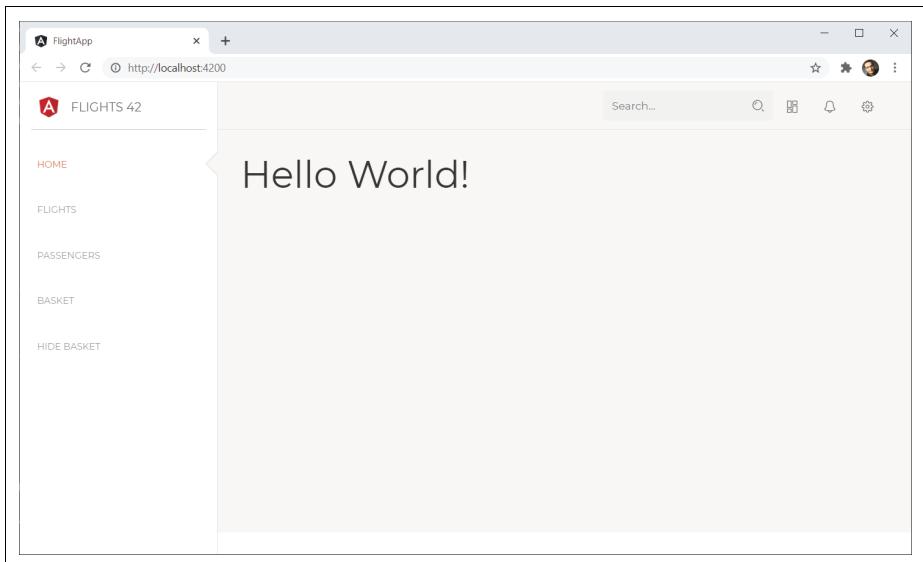


Abbildung 1-6: Anwendung mit Style-Bibliothek

Links sieht man die generierte `SideBarComponent` und im oberen Bereich die ebenfalls generierte `NavBarComponent`. Sämtliche Links sind derzeit noch Dummies – aber das wird sich im Laufe des Buchs noch ändern.



### ng add

Der von der Angular CLI gebotene Befehl `ng add` installiert ein npm-Paket und führt danach ein Skript aus, das das Paket in der Angular-Anwendung einrichtet.

Stattdessen könnte man auch mit `npm install` das Paket installieren und das Einrichten händisch übernehmen. Im Fall von Bootstrap müsste man dazu zunächst `npm install bootstrap` anstoßen und danach die von Bootstrap gebotenen Styles manuell in die `angular.json` eintragen. Danach müssten Sie auf dieselbe Weise die Styles des verwendeten Bootstrap-Themes referenzieren.

## Alternativen zu Bootstrap

Sie finden sowohl im Open-Source-Umfeld als auch im kommerziellen Bereich zahlreiche Implementierungen von Designsystemen und Komponentenbibliotheken, die sich als Alternative zu Bootstrap einsetzen lassen. Wir möchten hier auf ein paar, auf die wir in unserer Praxis immer wieder stoßen, hinweisen:

### *Angular Material (<https://material.angular.io>)*

Die offizielle und quelloffene Implementierung von Googles Material Design für Angular. Es handelt sich dabei um jene Implementierung, die Google für seine über 2.600 auf Angular basierenden Anwendungen nutzt.

### *Clarity Design System (<https://clarity.design/>)*

Immer mehr Unternehmen veröffentlichen die Umsetzungen ihrer Designsysteme. Hierzu zählt auch der bekannte Virtualisierungsriese VMWare, der sein Clarity Design System für Angular implementiert hat.

### *PrimeNG (<https://www.primefaces.org/primeng/>)*

Mit PrimeNG bietet der Hersteller der im Java-Umfeld bekannten PrimeFaces UI Frameworks eine umfangreiche und freie Komponentenbibliothek für Angular. Als Ergänzung werden kommerzielle Themes und Enterprise-Support geboten.

### *DevExtreme (<https://js.devexpress.com/>)*

Die Firma DevExpress dürfte dem einen oder anderen ein Begriff sein, zumal sie seit mehreren Jahrzehnten für verschiedene Plattformen ausgefeilte Komponenten anbietet. Ihr kommerzielles Produkt DevExtreme adressiert unter anderem Angular-Anwendungen. Es beinhaltet die üblichen Standardsteuerelemente und zeichnet sich durch ein äußerst mächtiges DataGrid aus. Aber auch ein sehr mächtiges Tree- und Charts-Control findet man hier.

### *KendoUI (<https://www.telerik.com/kendo-ui>)*

Mit KendoUI ist es ähnlich wie mit DevExtreme: Hinter diesem kommerziellen Produkt steht mit Telerik ein seit Jahrzehnten etablierter Anbieter von Steuerelementen. KendoUI bietet neben den üblichen Steuerelementen auch ein sehr mächtiges Grid-, Tree- und Chart-Control.

*ag-Grid (<https://www.ag-grid.com>)*

Wer mit seinem zum Beispiel mit Bootstrap umgesetzten Designsystem zufrieden ist und nur zusätzlich ein mächtiges Grid benötigt, wird bei ag-Grid fündig. Diese kommerzielle Lösung fokussiert sich darauf, »das beste JavaScript Grid der Welt« bereitzustellen – wie es sein Hersteller ausdrückt.

## Zusammenfassung

Das Projekt-Setup bei modernen JavaScript-Anwendungen fällt in der Regel recht komplex aus. Sie müssen Compiler konfigurieren, Werkzeuge für das Testing installieren und sich um einen Build kümmern, der zahlreiche Optimierungen für den Produktiveinsatz durchführt.

Die Angular CLI nimmt Ihnen viele dieser Aufgaben ab und generiert ein professionelles Setup, das diverse Werkzeuge zur Entwicklung mit Angular einrichtet. Es genügt ein einfaches `ng new`, und schon können Sie loslegen. Wie bei jedem generierten Projekt-Setup müssen Sie sich jedoch ein wenig Zeit nehmen, um sich mit den generierten Dateien vertraut zu machen. Vor allem die generierte `AppComponent` und das generierte `AppModule` werden Sie häufig anpassen müssen. Die zahlreichen generierten Konfigurationsdateien können Sie vorerst glücklicherweise als Black Box betrachten und bei Bedarf die eine oder andere Einstellung nachschlagen.



# Erste Schritte mit TypeScript

Angular-Anwendungen werden mit der Sprache TypeScript geschrieben. Dabei handelt es sich um eine typsichere Obermenge von JavaScript, die ein Compiler in handelsübliches JavaScript übersetzt.

Dieses Kapitel geht auf die Grundlagen von TypeScript ein und stellt die wichtigsten Sprachelemente daraus vor. Wir gehen davon aus, dass Sie bereits Erfahrung mit grundlegenden Programmierkonzepten haben und JavaScript zumindest im Überblick kennen. All jene, die sich schon mit TypeScript auseinandergesetzt haben, können dieses Kapitel getrost überspringen.

Wir haben auch die Erfahrung gemacht, dass sich C#- und Java-Entwickler sehr schnell in der Sprache TypeScript zurechtfinden. Viele Konzepte sind gleich. Falls Sie zu dieser Gruppe gehören, möchten Sie vielleicht dieses Kapitel auch vorerst überspringen und später bei Bedarf hier nachschlagen.

## Motivation

Im Jahr 2015 war es so weit: Das lang ersehnte ECMAScript 6, das den offiziellen Namen ECMAScript 2015 bekam, wurde zum Standard erklärt. Damit hat das Konsortium, das sich um die Weiterentwicklung von JavaScript kümmert, eine Obermenge zum bis dahin vorherrschenden JavaScript-Standard ECMAScript 5 geschaffen und neue Sprachelemente eingeführt. Bei diesen handelt es sich um Sprachelemente, die andere Sprachen schon länger anbieten und die manche im JavaScript-Standard schmerzlich vermissten. Beispiele dafür sind Klassen, Module oder Lambda-Ausdrücke, die das verkürzte Formulieren von Funktionen erlauben.

Seit jenem Zeitpunkt kommt es jährlich zu Erweiterungen des Standards. Die Sprache TypeScript, die bei Microsoft von Anders Hejlsberg – dem Vater von Delphi und C# – entwickelt wurde, setzt hier noch einen drauf: Sie versteht sich als Obermenge von ECMAScript und bietet bereits Sprachelemente, die für spätere Versionen geplant sind. Der wohl größte Mehrwert gegenüber ECMAScript ist jedoch ihr

statisches Typsystem. Es unterstützt den Programmierer beim frühzeitigen Erkennen von Fehlern, beim Refactoring von Quellcode sowie bei der Codevervollständigung.

Damit TypeScript im Browser ausgeführt werden kann, überführt es ein Compiler in handelsübliches ECMAScript. Projekte, die mit der Angular CLI erzeugt wurden, kompilieren standardmäßig nach ECMAScript 2015. Manche Menschen bevorzugen den Begriff *transpilieren*, da von einer Hochsprache in eine andere übersetzt wird.

## Mit TypeScript starten

Nachdem wir in Kapitel 1 mit dem Projekt-Setup den Grundstein für erste Schritte in TypeScript gelegt haben, gehen wir in diesem Abschnitt auf einige grundlegende Sprachelemente ein.

### Hallo Welt!

Als Basis für die ersten Schritte mit TypeScript kommt das Projekt-Setup aus dem vorigen Kapitel zum Einsatz. Um die Übersicht nicht zu verlieren, legen wir für unsere TypeScript-Experimente einen Ordner *ts* unter *src/app* an. Darin erzeugen wir eine neue Datei *demo.ts*. Weil es so ein schöner Brauch ist, gibt diese Datei erst mal *Hallo Welt!* auf der Entwicklerkonsole des Browsers aus:

```
// src/app/ts/demo.ts
console.debug('Hallo Welt!');
```

Diese Datei müssen Sie anschließend in Ihrer *main.ts* referenzieren, damit sie beim Programmstart ausgeführt wird. Fügen Sie dazu in der ersten Zeile die folgende `import`-Anweisung ein:

```
// src/main.ts
import './app/ts/demo';
[...]
```

Die Dateiendung *.ts* wird hier ganz bewusst weggelassen. Alle restlichen Einträge in der Datei *main.ts* kümmern sich, wie im letzten Kapitel besprochen, um das Bootstrapping der Angular-Anwendung.

Um das Projekt zu testen, starten Sie im Hauptordner des Projekts den Entwicklungswebserver:

```
ng serve -o
```

Danach sollten Sie unter *http://localhost:4200* Ihre Anwendung sehen. Öffnet man die Entwicklungswerkzeuge (*F5* oder *Strg+Umschalt+I*) und wechselt man auf die Konsole, sollte sich dort die Ausgabe *Hallo Welt!* finden (siehe Abbildung 2-1).

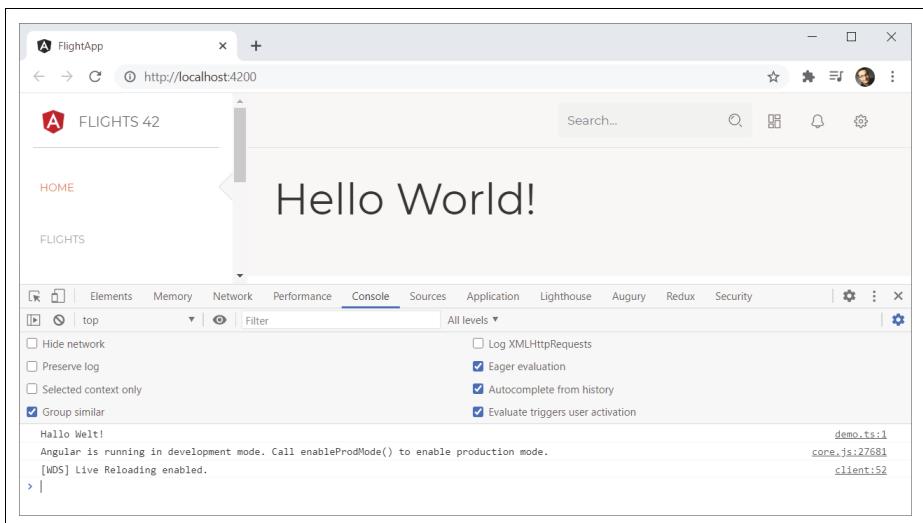
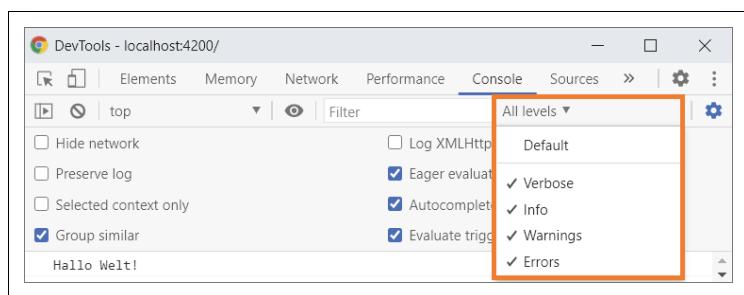


Abbildung 2-1: Hallo Welt auf der Konsole



Bitte beachten Sie, dass Chrome standardmäßig Debug-Ausgaben auf der Konsole unterdrückt. Um sie einzublenden, müssen Sie das Level *Verbose* aktivieren:



## Variablen deklarieren

Zum Deklarieren von Variablen bietet sich in TypeScript das mit ECMAScript 2015 eingeführte Schlüsselwort `let` an. Den gewünschten Datentyp geben Sie hinter dem Variablennamen an, und auf Wunsch kann auch ein initialer Wert zugewiesen werden:

```
let message: string = 'Hallo Welt!';
```

Der TypeScript-Compiler verhindert, dass ein Wert eines anderen Datentyps dieser Variablen zugewiesen wird:

```
// message = 42; // FEHLER
```



Standardmäßig erzeugt der TypeScript-Compiler selbst beim Auftreten von Kompilierungsfehlern JavaScript-Code. In manchen Fällen ist dieser sogar lauffähig. Selbstredend sollten sämtliche Kompilierungsfehler trotzdem behoben werden.

Ein explizites Angeben von Typen ist jedoch vielerorts gar nicht notwendig, zumal TypeScript den Typ durch Schlussfolgerungen herzuleiten versucht. Hierbei ist auch von Typerleitungen oder Typinferenz die Rede. Im betrachteten Fall erkennt TypeScript beispielsweise durch den zugewiesenen Initialwert, dass es sich um einen String handelt:

```
let message2 = 'Hallo Welt!';
```

Einige Entwicklungsumgebungen geben sogar Warnungen aus, wenn Sie unnötigerweise den Typ einer Variablen angeben. Damit möchte man die Schreibweise in der Community vereinheitlichen und Code leichter lesbar gestalten.

Um dies zu verhindern, können Variablen explizit mit `any` typisiert werden. Dieser Datentyp ahmt das übliche Verhalten von JavaScript nach und erlaubt das Zuweisen aller möglichen Werte:

```
let message3: any = 'Hallo Welt!';
message3 = 42; // OK
```

Wie die meisten anderen Sprachen erlaubt auch TypeScript die Deklaration von Konstanten, deren Werte sich im Nachhinein nicht verändern lassen. Hierzu kommt das Schlüsselwort `const` zum Einsatz, das ebenfalls mit ECMAScript 2015 eingeführt wurde:

```
const message4 = 'Hallo Welt!';
// message4 = 'Guten Tag'; // FEHLER
```

Es gehört in der Welt von TypeScript mittlerweile zum guten Ton, so viele Variablen wie möglich mit `const` zu deklarieren. Bei Änderungen müssten somit neue Konstanten erzeugt werden:

```
const message5 = 'Hallo';
const message6 = message5 + ' Welt';
```

Die Idee dahinter ist, dass die Nutzung von Konstanten zu besser lesbarem Code führt, unter anderem weil wir beim Ändern von Werten gezwungen sind, neue Konstantennamen zu vergeben.



Als Alternative zum Schlüsselwort `let` unterstützt TypeScript auch das von JavaScript bekannte Schlüsselwort `var`. Auf den ersten Blick führen beide Varianten zum selben Ergebnis. Allerdings gelten mit `var` deklarierte Variablen aus historischen Gründen immer ab dem Beginn der aktuellen Funktion – egal wo sie deklariert werden. Mit `let` deklarierte Variablen entsprechen hingegen stärker dem Prinzip der geringsten Überraschung, zumal sie ab der Deklaration und nur für den aktuellen Block gelten:

```

function demo(a: number, b: number): number {
    // y existiert ab hier

    if (a > b) {
        let x = -1;
        // x existiert ab hier und nur in diesem Block.

        return x;
    }
    else {
        var y = 1;
        return y;
    }
}

```

## Ausgewählte Datentypen in TypeScript

Dieser Abschnitt präsentiert die für den Umgang mit Angular wichtigsten Datentypen, die TypeScript zu bieten hat.

### **number**

Der Datentyp `number` entspricht dem gleichnamigen Datentyp aus JavaScript. Es handelt sich dabei prinzipiell um eine Fließkommazahl doppelter Genauigkeit (in vielen Sprachen unter `double` bekannt). Ganzzahlen im Wertebereich zwischen  $-2^{53}$  und  $2^{53}$  (exklusive  $-2^{53}$  und  $2^{53}$ ) werden jedoch als »sichere« Integer (*Safe Integer*) ohne Genauigkeitsverluste dargestellt:

```

const i: number = 42;      // Safe Integer
const d: number = 0.815;   // Fließkommazahl

```

Um einen Wert in eine `number` umzuwandeln, die JavaScript intern durch einen Integer repräsentiert, kommt die Funktion `parseInt` zum Einsatz. Für Fließkommazahlen steht hingegen `parseFloat` zur Verfügung. Möchte man prüfen, ob ein Wert eine gültige Zahl ist, nutzt eine Anwendung `isNaN`, wobei `NaN` für *Not a Number* steht:

```

const i2: number = parseInt('42', 10);
// String 42 als Integer.
// 10 ist die Basis (Dezimalsystem).

const d2: number = parseFloat('0.815');
// String 0.815 als Fließkommazahl.

const str: any = 'ein String';

if (isNaN(str)) {
    console.debug('Ein String ist keine Zahl!');
}

```

Die explizite Angabe einer Basis bei `parseInt` wird empfohlen, da in der Vergangenheit unterschiedliche Browser hierfür verschiedene Standardwerte herangezogen haben.

Für dieses Experiment musste `str` als `any` deklariert werden. Ansonsten hätte das Typsystem den Aufruf `isNaN(str)` verhindert, zumal von vornherein klar ist, dass eine String-Variablen keine Zahl ist.

Der Wert `NaN` kommt auch zum Einsatz, wenn das Ergebnis einer mathematischen Funktion nicht definiert ist. Ein Beispiel dafür ist die Division durch null (0).

## string

Ein `string` ist eine Zeichenkette, die durch ein Objekt der Klasse `String` repräsentiert wird. Als Begrenzungszeichen kommen wahlweise einfache oder doppelte Anführungszeichen zum Einsatz. Auch Backticks sind mittlerweile möglich. Sie unterstützen Zeilenschaltungen sowie Platzhalter, die JavaScript-Ausdrücke in den String aufnehmen:

```
const name1: string = 'Max';
const name2: string = 'Max';
const name3: string =
` ${name2} Mustermann`;
```



Sie können anstatt des Datentyps `string` auch den Klassennamen `String` angeben. Beide Schreibweisen sind bedeutungsgleich.

Zum Verketten von Strings kommt ein Pluszeichen zum Einsatz:

```
const name4 = name2 + ' Muster';
```

Daneben bietet die Klasse `String` die üblichen Methoden, wie `substring`, `charAt` oder `length`. Eine gute Übersicht dazu finden Sie unter anderem im Mozilla Developer Network ([https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/String](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String)).

## boolean

Ein `boolean` repräsentiert einen Wahrheitswert (`true` oder `false`):

```
const ok: boolean = false;
```

Eine interessante Eigenheit von JavaScript und somit auch von TypeScript ist, dass sämtliche Datentypen als `boolean` behandelt werden können. Hierbei ist auch von *truthy* und *falsy* die Rede. Werte, die *truthy* sind, werden als `true` interpretiert, Werte, die *falsy* sind, als `false`.

*Falsy* sind neben `false` die Werte `0`, `''` (Leerstring), `null`, `undefined` und `NaN` (*Not a Number*). Alle anderen Werte sind *truthy*:

```
const firstName = null;
if (!firstName) {
    console.debug('firstName is falsy');
}
```

## Arrays

Arrays nehmen mehrere Werte eines Typs auf und sind auch dynamisch erweiterbar. Das folgende Beispiel deklariert ein String-Array mit zwei Namen. Zwei weitere fügt es mit `push` ein. Danach iteriert es mit der `for-of`-Schleife (sie wurde mit ECMAScript 2015 eingeführt) über sämtliche Einträge und gibt sie aus:

```
const namen: string[] = ['Max', 'Susi'];
// gleichbedeutende Alternative: const namen: Array<string> = ['Max', 'Susi'];
namen.push('Rainer'); // hinzufügen
namen.push('Anna'); // hinzufügen

for(const entry of namen) {
    console.debug(entry);
}
```

Für eine vollständige Übersicht über alle Array-Methoden verweisen wir auch hier auf die Dokumentation des Mozilla-Projekts: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array).

## any

Den Variablen des Typs `any` darf eine Anwendung alle möglichen Werte zuweisen. Somit simuliert der Typ `any` das Standardverhalten der dynamisch typisierten Sprache JavaScript:

```
const otherName: any = 'Max Muster';
otherName = 42;
```

## unknown

Der Datentyp `any` ist nicht besonders gern gesehen, zumal er die Prinzipien der Typisierung untergräbt. Als Alternative bietet sich jedoch häufig der Datentyp `unknown` an.

Wie Variablen vom Typ `any` kann eine Variable vom Typ `unknown` sämtliche Werte aufnehmen:

```
let value: unknown = 'Hallo Welt!';
value = 100;
```

Bevor Sie allerdings mit dem Wert solch einer Variablen arbeiten dürfen, müssen Sie sich vergewissern, dass sie den richtigen Typ aufweist:

```
if (typeof value === 'number') {
    const result = value + 10;
    console.debug(result);
}
```

Somit ist sichergestellt, dass die Anwendung die Variable zur Laufzeit korrekt verwendet.

## Function

Variablen des Typs `Function` verweisen auf eine Funktion:

```
const f: Function = function() {
    console.debug('Hallo Welt!');
}

f(); // Aufruf von f
```

Alternativ dazu kann ein Funktionstyp, der auch die Übergabeparameter und den Rückgabewert festlegt, definiert werden:

```
type MathFn = (a: number, b: number) => number;

const func: MathFn = function(a: number, b: number) {
    return a + b;
}

console.debug(func(1,2));
```

Gegenüber der bloßen Verwendung des Typs `Function` ist diese Schreibweise typsicher: TypeScript stellt sicher, dass die Variable `func` vom Typ `MathFn` nur auf Funktionen mit der angegebenen Signatur verweist.

## Klassen und Interfaces

Eigene Datentypen lassen sich unter anderem mit Interfaces und Klassen definieren. Die nächsten Abschnitte gehen darauf ein.



JavaScript bringt schon einige Klassen mit, die sich über TypeScript natürlich auch nutzen lassen. Beispiele dafür sind die Klassen `String` zur Verwaltung von Zeichenketten, `Date` zum Repräsentieren von Datumswerten oder `Math` mit mathematischen Funktionen. Einen guten Überblick, der einen offiziellen Charakter hat, finden Sie im Mozilla Developer Network ([https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects)).

## Ein erstes Objekt samt Modul

Objekte sind ein zentrales Element in JavaScript und in TypeScript, aber auch in Angular. Die meisten Konzepte lassen sich durch Objekte beschreiben. Beispiele dafür sind Daten vom Server, Formulare und Eingabefelder oder Services mit wiederverwendbaren Programmteilen.

Eine einfache Möglichkeit, um die Struktur von Objekten festzulegen, ist die Nutzung von Interfaces. Sie definieren die benötigten Felder samt ihren Datentypen. Zur Veranschaulichung definiert Beispiel 2-1 ein Interface für Flüge.

### *Beispiel 2-1: Interface für einen Flug*

```
// src/app/flight.ts
export interface Flight {
    id: number;
    from: string;
    to: string;
    date: string;
    delayed?: boolean;
}
```

Dieses erste einfache Interface gibt lediglich Eigenschaften vor. Bitte beachten Sie das Fragezeichen hinter `delayed`. Es legt fest, dass diese Eigenschaft optional ist. Der Abschnitt »Interfaces und Vererbung« auf Seite 53 zeigt darüber auch Interfaces, die Methoden vorgeben.

Wie der Kommentar am Beginn des Listings andeutet, wurde das betrachtete Interface in einer Datei `src/app/flight.ts` eingerichtet. Das ist insofern von Bedeutung, als ab ECMAScript 2015 jede Datei ein eigenes Modul darstellt. Dieses Modul ist von anderen Modulen abgeschottet und hat seinen eigenen Namensraum. Das bedeutet, dass alles, was eine Datei definiert, zunächst nur innerhalb dieser Datei existiert. Das beugt Namenskonflikten vor, zum Beispiel in Fällen, in denen mehrere Dateien etwas mit dem Namen `Flight` definieren.

Beachten Sie, dass wir die Datei `flight.ts` unter `src/app/` und nicht wie die anderen Dateien dieses Kapitels unter `src/app/ts` ablegen. Der Grund dahinter ist, dass wir diese Datei in fast allen Kapiteln dieses Buchs verwenden und deswegen einen allgemeineren Ordner dafür definieren wollten.

Um anderen Modulen Konstrukte wie Interfaces, Klassen oder auch nur Variablen zur Verfügung zu stellen, kommt das Schlüsselwort `export` zum Einsatz. Solche Konstrukte können in anderen Dateien bei Bedarf importiert werden. Ein Beispiel dafür findet sich in Beispiel 2-2.

### *Beispiel 2-2: Flugobjekt erzeugen*

```
// src/app/ts/demo.ts
import { Flight } from '../flight';

[...]

const flight: Flight = {
    id: 1,
    from: 'Graz',
    to: 'Hamburg',
    date: '2018-12-24T17:00:00.0001:00'
}

flight.from = 'GRZ';
flight.to = 'HAM';

console.debug('from', flight.from);
console.debug('flight', flight);
```



ECMAScript sowie TypeScript bieten eine vereinfachte Schreibweise für Fälle, in denen Sie eine Variable einer gleichnamigen Objekteigenschaft zuweisen wollen. Beispielsweise lässt sich anstatt

```
const id = 1;
const from = 'Graz';
const to = 'Hamburg';
const date = '2018-12-24T17:00:00.0001:00';

const flight: Flight = {
    id: id,
    from: from,
    to: to,
    date: date
};
```

auch die folgende Form verwenden:

```
const id = 1;
const from = 'Graz';
const to = 'Hamburg';
const date = '2018-12-24T17:00:00.0001:00';

const flight: Flight = { id, from, to, date };
```

Wir können zum Ausprobieren der einzelnen Sprachelemente die bestehende Datei *demo.ts* erweitern. Um den Code aus den anderen Beispielen nicht wiederholen zu müssen, nutzen wir das Auslassungszeichen [...].

Es handelt sich dabei um die Datei *src/app/ts/demo.ts*, die in der ersten Zeile das *Flight*-Interface aus der Datei *flight.ts* importiert. Den Dateinamen der *flight.ts* gibt es dazu relativ an, und eine Dateiendung lässt es ganz bewusst weg. Das ist auch gut so, denn das, was beim Kompilieren eine *.ts*-Datei ist, wird bei der Ausführung eine eigene *.js*-Datei oder sogar nur ein kleiner Teil eines JavaScript-Bundles sein.



Bitte beachten Sie auch, dass `import`-Anweisungen typischerweise am Anfang der Datei stehen.

Das betrachtete Beispiel definiert eine Variable vom Typ `Flight` und weist mit einem sogenannten Objektliteral ein neues Objekt zu. Der TypeScript-Compiler stellt hier sicher, dass sämtliche Vorgaben des Interface berücksichtigt sind. Das Weglassen einer der vier vorgegebenen Eigenschaften würde somit zu einem Fehler führen.

Eine etwas genauere Betrachtung verlangt die Eigenschaft `date`. Sie erhält einen String mit einem ISO-Datum. JavaScript hat zwar auch eine Klasse `Date`, um Datumsdaten mit einem Objekt zu beschreiben, aber diese Klasse lässt sich nicht per JSON zwischen Server und Client übertragen. Das JSON-Format sieht dafür schlicht und ergreifend keine Repräsentation vor. Deswegen hat sich die Community auf die Nutzung von Strings mit dem ISO-Format geeinigt, das sowohl ein Datum und eine

Uhrzeit als auch ein Zeitzonenoffset beinhalten kann. Das Offset wird hier durch die Endung +01:00 für die mitteleuropäische Winterzeit ausgedrückt.

## Auto-Importe mit Visual Studio Code

Glücklicherweise muss man die Importe in der Regel nicht manuell einfügen. Visual Studio Code bietet nämlich hierfür eine Auto-Import-Funktion. In vielen Fällen schlägt die Entwicklungsumgebung schon beim Tippen mögliche Importe vor. Ist das nicht der Fall, bewegt man den Cursor über das zu importierende Konstrukt und klickt auf das erscheinende Symbol (siehe Icon in Zeile 55 in Abbildung 2-2).

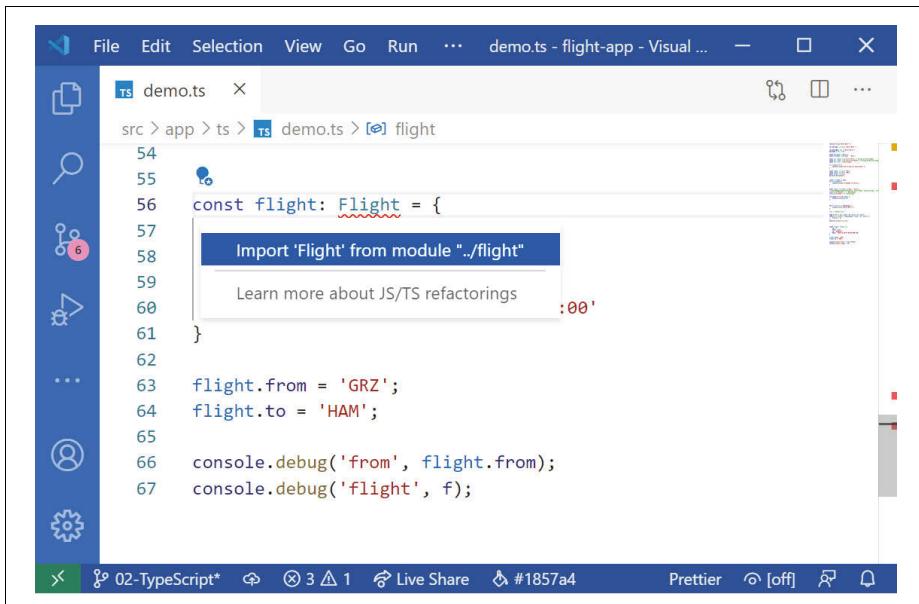


Abbildung 2-2: Automatische Importe in Visual Studio Code

Die Tastenkombination Strg+. bewirkt dasselbe. Leider versagt dieser Automatismus ab und an. In diesen Fällen muss man den `import` manuell hinzufügen.

## Klassen

Im Gegensatz zu Interfaces können Klassen neben Eigenschaften auch Funktionen für die beschriebenen Objekte festlegen. Hierbei ist ebenfalls von Methoden die Rede. Ein Beispiel dafür findet sich in Beispiel 2-3. Es definiert eine Klasse Flight Manager zum Verwalten von Flügen.

*Beispiel 2-3: Klasse zum Verwalten von Flügen*

```
// src/app/ts/flight-manager.ts
import { Flight } from './flight';
```

```

export class FlightManager {

    private cache: Flight[];

    constructor(cache: Flight[]) {
        this.cache = cache;
    }

    search(from: string, to: string): Flight[] {
        const result = new Array<Flight>();
        for (let f of this.cache) {
            if (f.from === from && f.to === to) {
                result.push(f);
            }
        }
        return result;
    }
}

```



Neben den aus anderen Sprachen bekannten Operatoren `==` und `!=` zum Vergleichen von Werten bieten JavaScript und somit TypeScript auch die Operatoren `===` und `!==`. Letztere beziehen den Datentyp ebenfalls in die Gleichheitsprüfung ein:

```

const b1 = '1' == 1; // true
const b2 = '2' === 2; // false

```

Generell sind deswegen `===` und `!==` zu bevorzugen.

Diese Klasse weist eine Eigenschaft `cache` auf, die vom Typ `Flight`-`Array` (`Flight[]`) ist und somit mehrere Flüge aufnehmen kann.

Der Konstruktor ist eine Methode, die TypeScript beim Erzeugen von Objekten dieser Klasse aufruft. Er nimmt einen Initialwert für den Cache entgegen. Auf das Feld `cache` greift er über das Schlüsselwort `this` zu, das das aktuelle Objekt repräsentiert. Anders als bei vielen ähnlichen Sprachen ist der Einsatz von `this` in TypeScript verpflichtend.

Die Methode `search` sucht anhand von übergebenen Kriterien nach Flügen im Cache und liefert ein Array mit den Treffern zurück. Dabei fällt auf, dass sie neben der Schreibweise `Flight[]` auch die Schreibweise `Array<Flight>` verwendet. Beides hat in TypeScript die gleiche Bedeutung.



Da TypeScript versucht, Typen über Schlussfolgerungen herzuleiten, könnte in Beispiel 2-3 der Rückgabewert der Methode `search` auch weggelassen werden. In diesem Fall würde TypeScript durch Analyse der Verwendung von `return` auf den Datentyp `Flight[]` schließen. Wer sich mit expliziten Typangaben besser fühlt, kann diese natürlich jederzeit verwenden.

Damit die Eigenschaft `cache` nur innerhalb der Klasse sichtbar ist, wurde sie mit dem Access-Modifier `private` versehen. Die Methode `search` hat hingegen keinen

Access-Modifier und ist somit öffentlich zugänglich. Insgesamt bietet TypeScript die folgenden Access-Modifier:

*public*

Jeder hat Zugriff. Es handelt sich hierbei um den Standardwert, d.h., Eigen-schaften ohne Access-Modifier sind immer `public`.

*protected*

Nur die Klasse selbst und davon abgeleitete Klassen dürfen zugreifen. Details zur Ableitung von Klassen finden Sie weiter unten unter »Interfaces und Ver-erbung« auf Seite 53.

*private*

Nur die Klasse selbst hat Zugriff.

*readonly*

Wird gemeinsam mit `public`, `protected` und `private` verwendet. Legt fest, dass der Wert nach seiner Initialisierung nicht mehr geändert werden darf. Die Ini-tialisierung findet entweder durch Zuweisung eines Standardwerts statt oder im Konstruktor.



Konstruktoren kommen häufig zum Initialisieren von Eigenschaften zum Einsatz. Im betrachteten Fall ist die Eigenschaft `cache` ein Bei-spiel dafür:

```
private cache: Flight[];  
  
constructor(cache: Flight[]) {  
    this.cache = cache;  
}
```

Dieses Fragment weist das Konstruktorargument `cache` der gleichna-migen privaten Eigenschaft zu. Somit kommt das Wort `cache` hier viermal vor. Um dieses wortreiche Unterfangen abzukürzen, bietet TypeScript ein wenig syntaktischen Zucker:

```
constructor(private cache: Flight[]) {  
}
```

Hierzu müssen Sie lediglich das Konstruktorargument mit einem Ac-cess-Modifier wie `private` oder `public` versehen. In diesem Fall richtet TypeScript sowohl ein Konstruktorargument als auch eine gleichna-mige Eigenschaft ein und weist Ersteres Letzterem zu. Die Semantik entspricht somit der des zuvor betrachteten ausführlichen Konstrukts.

Ein Beispiel für die Nutzung der Klasse `FlightService` findet sich in Beispiel 2-4. Es importiert sowohl `Flight` als auch `FlightManager` und erzeugt danach mit einem Objektliteral ein `Array<Flight>`, das als Cache fungiert:

*Beispiel 2-4: Der FlightService nimmt ein Array entgegen.*

```
// src/app/ts/demo.ts  
import { Flight } from '../flight';  
import { FlightManager } from './flight-manager';
```

```

let flights: Array<Flight> = [
  {
    id: 17,
    from: 'Graz',
    to: 'Hamburg',
    date: '2022-02-27T17:00+01:00'
  },
  {
    id: 18,
    from: 'Graz',
    to: 'Hamburg',
    date: '2022-02-27T18:00+01:00'
  },
  {
    id: 19,
    from: 'Graz',
    to: 'Mallorca',
    date: '2022-02-27T19:00+01:00'
  },
  {
    id: 20,
    from: 'Graz',
    to: 'Hamburg',
    date: '2022-02-27T20:00+01:00'
  }
];
let fm = new FlightManager(flights);
let result1 = fm.search('Graz', 'Hamburg');

for (let f of result1) {
  console.debug('flight', f);
}

```

Mit dem Schlüsselwort `new` erzeugt das betrachtete Beispiel ein neues Objekt nach der Vorlage des `FlightManager` und übergibt dabei das Array `flights` an dessen Konstruktor. Anschließend sucht es mit `search` nach Flügen von Graz nach Hamburg und gibt das erhaltene Ergebnis aus. Dazu iteriert es das Ergebnis mit einer `for-of`-Schleife.

## Funktionen und Lambda-Ausdrücke

Logiken müssen bei JavaScript nicht in Form von Methoden innerhalb von Klassen gekapselt sein. Vielmehr unterstützt die Sprache des Webs auch allein stehende Funktionen. Wie Beispiel 2-5 veranschaulicht, kommt dazu das Schlüsselwort `function` zum Einsatz. Um anzudeuten, dass eine Funktion keine Werte zurückliefert, verwendet sie den Datentyp `void`.

### *Beispiel 2-5: Eine einfache Funktion*

```

// src/app/ts/demo.ts
function showFlight(f: Flight): void {
  console.debug('--- Flight ---');
}

```

```

        console.debug('id', f.id);
        console.debug('date', f.from);
        console.debug('date', f.to);
        console.debug('date', f.date);
    }
}

```

Ein Beispiel für die Nutzung dieser Funktion findet sich in Beispiel 2-6.

#### *Beispiel 2-6: Funktionen nutzen*

```

const anotherFlight: Flight = {
    id: 1,
    from: 'Graz',
    to: 'Hamburg',
    date: '2018-12-24T17:00:00.00+01:00'
}

showFlight(anotherFlight);

```

So richtig spannend wird es, wenn es um den Einsatz anonymer Funktionen geht. Damit sind Funktionen gemeint, die keinen Namen haben, jedoch an Ort und Stelle verwendet werden. Beispiel 2-7 spendiert zum Beispiel unserem FlightManager eine Methode search2, die die Array-Methode filter nutzt.

#### *Beispiel 2-7: Anonyme Funktion*

```

// src/app/ts/flight-manager.ts
import { Flight } from '../flight';

export class FlightManager {

    constructor(private cache: Flight[]) {
    }

    search2(from: string, to: string): Flight[] {
        const result: Flight[] = this.cache.filter(function (f: Flight) {
            return f.from === from && f.to === to;
        });
        return result;
    }

    [...]
}

```

Die Funktion filter hat die Aufgabe, ein Array zu filtern, und nimmt die Filterkriterien in Form einer *anonymen Funktion* entgegen.

Sie ruft die übergebene anonyme Funktion für jeden Eintrag im Array auf. Liefert diese true, nimmt filter den jeweiligen Eintrag in die Ergebnisliste auf; ansonsten wird er außen vor gelassen. Das so ermittelte Ergebnis liefert filter zurück.

Eine verkürzte Schreibweise dafür bieten die mit ECMAScript 2015 eingeführten Lambda-Ausdrücke. Sie heißen offiziell *Arrow-Funktionen*, weil in ihnen der Pfeil-

operator (`=>`) genutzt wird. *Lambda-Ausdruck* ist hingegen der sprachneutrale Name, der sich eingebürgert hat. Wie Beispiel 2-8 zeigt, ähnelt ein solcher Ausdruck der Nutzung *anonymer Funktionen*:

*Beispiel 2-8: Lambda-Ausdruck*

```
// src/app/ts/flight-manager.ts
search3(from: string, to: string): Flight[] {
    return this.cache.filter((f: Flight) => {
        return f.from === from && f.to === to;
    });
}
```

Lediglich das Schlüsselwort `function` wird weggelassen, und als Trennzeichen zwischen Parameterliste und Body der Funktion kommt der Pfeiloperator (`=>`) zum Einsatz. Gelesen wird dieser Operator als *goes to*, was sich im Deutschen mit *wird abgebildet auf* übersetzen lässt. Sprachlich wurde das der Mathematik entlehnt, wo Funktionen Eingaben auf Ausgaben *abbilden*.

Die Schreibweise von Lambda-Ausdrücken lässt sich in manchen Fällen drastisch verkürzen. Beispiel 2-9 lässt beispielsweise die runden Klammern um die Parameter weg. Das ist möglich, wenn nur ein Parameter, dessen Typ hergeleitet wird, zum Einsatz kommt.

*Beispiel 2-9: Kürzere Schreibweise für einen Lambda-Ausdruck*

```
// src/app/ts/flight-manager.ts
search3(from: string, to: string): Flight[] {
    return this.cache.filter(f => {
        return f.from === from && f.to === to;
    });
}
```

Beispiel 2-10 lässt sogar noch die geschweiften Klammern zur Begrenzung des Funktionskörpers sowie das Schlüsselwort `return` weg. Das ist möglich, wenn die Funktion nur aus einer einzigen Zeile besteht. In diesem Fall zieht ECMAScript 2015 das Ergebnis dieser Zeile als Rückgabewert heran.

*Beispiel 2-10: Noch kürzere Schreibweise für einen Lambda-Ausdruck*

```
// src/app/ts/flight-manager.ts
search3(from: string, to: string): Flight[] {
    return this.cache.filter(f => f.from === from && f.to === to);
}
```



Neben der verkürzten Schreibweise bringen Lambda-Ausdrücke noch eine weitere Annehmlichkeit mit sich: Sie binden den Wert von `this`. Das bedeutet, dass `this` innerhalb des Lambda-Ausdrucks auf dasselbe Objekt wie außerhalb des Lambda-Ausdrucks zeigt. Für Entwickler aus der Welt von Java oder .NET ist das ohnehin das erwartete Verhalten. Erfahrene JavaScript-Entwicklerinnen und -Entwickler wissen jedoch, dass der Aufrufer einer Funktion den Wert von `this` bestimmt.

Die Nutzung des Schlüsselworts `function` führt jedoch aus Gründen der Abwärtskompatibilität nach wie vor zu diesem etwas eigenwilligen Verhalten. Wer sich damit nicht belasten möchte, nutzt generell für Callback-Funktionen Lambda-Ausdrücke.

## Interfaces und Vererbung

Interfaces und Klassenvererbung kommen in objektorientierten Sprachen zum Einsatz, um Anwendungsteile austauschbar zu gestalten. Zwei Objekte, die dasselbe Interface implementieren, sind aus Sicht des Compilers gegeneinander austauschbar. Diesen Umstand nutzen Frameworks wie Angular, um die Flexibilität zu steigern und um Standardverhalten anzupassen.

### Interfaces

Um den Einsatz von Interfaces zur Schaffung austauschbarer Objekte zu veranschaulichen, greifen wir in Beispiel 2-11 das bereits verwendete *Flight*-Interface wieder auf:

Beispiel 2-11: Interface mit Methode

```
// src/app/flight.ts
export interface Flight {
  id: number;
  from: string;
  to: string;
  date: string;

  distance?: number;
  calcPrice?(): number;
}
```

Als Ergänzung bekommt `Flight` zwei weitere Einträge spendiert: `distance` gibt Auskunft über die Länge der Flugstrecke, und `calcPrice` ist eine Methode zum Berechnen des Flugpreises. Der Fragezeichenoperator (?) gibt an, dass diese beiden Erweiterungen optional sind.

Im Gegensatz zu Klassen definiert ein Interface für Methoden nur eine Signatur, jedoch keinen Körper mit einer gewünschten Logik. Das Bereitstellen des Körpers ist Aufgabe jener Klassen, die das Interface implementieren. Ein weiterer Unterschied zu Klassen ist, dass sämtliche Einträge eines Interface per definitionem öffentlich (`public`) sind.

Die Klasse `ScheduledFlight` in Beispiel 2-12 implementiert `Flight` explizit. Dazu listet sie dieses Interface in ihrer `implements`-Klausel auf. Das veranlasst den Compiler, zu prüfen, ob sich `ScheduledFlight` an die Vorgaben des Interface hält.

*Beispiel 2-12: Implementierung von Flight*

```
// src/app/ts/scheduled-flight.ts

import { Flight } from '../flight';

export class ScheduledFlight implements Flight {

    id: number = 0;
    from: string = '';
    to: string = '';
    date: string = '';

    distance: number = 0;

    calcPrice(): number {
        return this.distance / 3;
    }

}
```

Bitte beachten Sie hier, dass jede Eigenschaft bewusst einen Standardwert zugewiesen bekommt. Das hängt damit zusammen, dass die Angular CLI seit Version 12 standardmäßig den sogenannten Strict Mode verwendet. Unter anderem verpflichtet dieser TypeScript dazu, strengere Prüfungen durchzuführen. Außerdem verbietet dieser Modus die standardmäßige Verwendung von nicht initialisierten Variablen. Wir gehen weiter unten unter »Strikte Null-Prüfungen« auf Seite 66 darauf etwas näher ein.



Eine Klasse kann beliebig viele Interfaces implementieren. Diese sind kommagetrennt in der `implements`-Klausel anzugeben:

```
class ScheduledFlight implements Flight, Billable,
Discountable, Offer
{
    [...]
}
```

In diesem Fall muss die Klasse sämtliche Vorgaben aller Interfaces erfüllen.

Die Klasse `CharterFlight` aus Beispiel 2-13 implementiert ebenfalls das Interface `Flight`. Allerdings unterscheidet sich die Implementierung von `calcPrice` im Detail von jener in `ScheduledFlight`.

*Beispiel 2-13: Eine weitere Implementierung des Interface Flight*

```
// src/app/ts/charter-flight.ts

import { Flight } from '../flight';

export class CharterFlight implements Flight {

    id: number = 0;
    from: string = '';
    to: string = '';
```

```

date: string = '';
distance: number = 0;
calcPrice(): number {
    return this.distance / 2;
}
}

```

Dadurch, dass beide Klassen das Interface implementieren, können beide Variablen des Interface-Typs `Flight` zugewiesen werden. Somit kann ein `ScheduledFlight`-Objekt ein `CharterFlight`-Objekt ersetzen, wie Sie in Beispiel 2-14 sehen:

*Beispiel 2-14: Austauschen von Objekten*

```

// src/app/ts/demo.ts
import { Flight } from '../flight';
import { ScheduledFlight } from './scheduled-flight';
import { CharterFlight } from './charter-flight';

[...]

let oneMoreFlight: Flight = new ScheduledFlight();
oneMoreFlight.distance = 1000;

if (oneMoreFlight.calcPrice) {
    console.debug('Preis', oneMoreFlight.calcPrice());
}

oneMoreFlight = new CharterFlight();
// Ersetzen; dieselbe Variable zeigt nun
// auf einen CharterFlight.

oneMoreFlight.distance = 1000;

if (oneMoreFlight.calcPrice) {
    console.debug('Preis', oneMoreFlight.calcPrice()); // neuer Preis
}

```

Da `calcPrice` im Interface als optionale Methode definiert wurde, ist das `if`, das deren Vorhandensein prüft verpflichtend.

Sie können aber auch Code schreiben, der verschiedene Ausprägungen des `Flight`-Interface auf die gleiche Weise verwendet. Dies nennt man auch Polymorphie. Ein Beispiel dafür bietet Beispiel 2-15.

*Beispiel 2-15: Polymorphe Behandlung zweier Ausprägungen von Flight*

```

// src/app/ts/demo.ts
import { Flight } from '../flight';
import { ScheduledFlight } from './scheduled-flight';
import { CharterFlight } from './charter-flight';

[...]

const scheduledFlight: Flight = new ScheduledFlight();

```

```

scheduledFlight.distance = 1000;

const charterFlight: Flight = new CharterFlight();
charterFlight.distance = 1000;

const myFlights: Flight[] = [scheduledFlight, charterFlight];

for(const f of myFlights) {
  if (f.calcPrice) {
    console.debug('Preis', f.calcPrice());
  }
}

```

Das Array `flights` vom Typ `Flight[]` beinhaltet sowohl einen `ScheduledFlight` als auch einen `CharterFlight`. Die Schleife geht auch nicht auf die Unterschiede zwischen diesen beiden Ausprägungen ein, sondern nutzt die vom Interface vorgegebenen Funktionen zur Preisberechnung. Da `calcPrice`, wie oben erwähnt, eine optionale Methode ist, prüft die Schleife mit einem `if`, ob diese Methode existiert.



Interfaces verschwinden beim Kompilieren und existieren somit nicht mehr zur Laufzeit. Der Grund dafür ist, dass dynamische Sprachen wie JavaScript gänzlich ohne dieses Konstrukt auskommen.

Das hier gezeigte Beispiel funktioniert auch, wenn die Klassen das Interface *nicht* in ihrer `implements`-Klausel erwähnen. In diesem Fall prüft der Compiler jedoch bei der Klassendeklaration nicht, ob die Klasse das Interface implementiert. Allerdings findet eine ähnliche Prüfung bei der Zuweisung zur Variablen des Interface-Typs statt:

```
let scheduledFlight: Flight = new ScheduledFlight();
```

An dieser Stelle prüft der Compiler, ob der `ScheduledFlight` eine zum `Flight` kompatible Struktur aufweist. Ist dem nicht so, mahnt er die Zuweisung mit einem Fehler an. Man spricht hierbei auch von Duck-Typing: Was aussieht wie eine Ente, ist eine Ente.

## Klassenvererbung

Anstatt ein Interface zu implementieren, kann eine Klasse auch von einer Basisklasse erben. Im Zuge dessen übernimmt die erbende Klasse sämtliche Eigenschaften und Methoden von der Basisklasse. Basisklassen werden auch als *Superklassen* und erbende Klassen dementsprechend als *Subklassen* bezeichnet.

Als Beispiel für eine Basisklasse kommt in Beispiel 2-16 die Klasse `Person` zum Einsatz:

### *Beispiel 2-16: Basisklasse Person*

```
// src/app/ts/person.ts
```

```

export class Person {
    id: number = 0;
    firstName: string = '';
    lastName: string = '';

    fullName() {
        return this.firstName + ' ' + this.lastName;
    }
}

export class Passenger extends Person {
    passengerStatus: string = '';
}

export class Pilot extends Person {
    licenseNumber: string = '';
}

```

Diese Klasse unterscheidet sich zunächst nicht von herkömmlichen Klassen. Allerdings erben die Klassen `Passenger` und `Pilot` von ihr. Dazu erwähnen sie die Person in ihrer `extends`-Klausel. Genau das macht `Person` in diesem Kontext zur Basisklasse.



Ähnlich wie andere Mainstream-Sprachen erlaubt TypeScript keine Mehrfachvererbung. Eine Klasse kann also immer nur von einer anderen erben.

Durch die `extends`-Klausel übernehmen `Passenger` und `Pilot` sämtliche Felder und Methoden von `Person`. Zur Veranschaulichung ergänzen beide jeweils ein zusätzliches Feld. Ähnlich wie beim Einsatz von Interfaces lassen sich Instanzen von Subklassen zu Variablen der Superklasse zuweisen. Durch die Vererbung hat die Subklasse auf jeden Fall eine zur Superklasse kompatible Struktur.

Ein Beispiel dafür findet sich in Beispiel 2-17. Sowohl der neue `Passenger` als auch der neue `Pilot` werden hier einer Variablen vom Typ `Person` zugewiesen:

*Beispiel 2-17: Polymorphe Behandlung zweier Ausprägungen von Person*

```

// src/app/ts/demo.ts

import { Passenger, Person, Pilot } from './person';

[...]

const person1: Person = new Passenger();
person1.firstName = 'Max';
person1.lastName = 'Muster';

const person2: Person = new Pilot();
person2.firstName = 'Jens';
person2.lastName = 'Wolkenmeyer';

const isPerson = person1 instanceof Person; // true
const isPassenger = person1 instanceof Passenger; // true

```

```
const isPilot = person1 instanceof Pilot; // false

console.debug('isPerson', isPerson);
console.debug('isPassenger', isPassenger);
console.debug('isPilot', isPilot);
```

Beachtenswert ist in Beispiel 2-17 auch der Einsatz des `instanceof`-Operators. Er prüft, ob ein Objekt zu einem gegebenen Klassentyp kompatibel ist. Da es sich bei `person1` um einen Passagier handelt, ist sie natürlich kompatibel zur Klasse `Passenger` – oder anders ausgedrückt: Sie lässt sich den Variablen von `Passenger` zuweisen, auch wenn hierzu gegebenenfalls eine Type Assertion notwendig ist – siehe dazu den folgenden Abschnitt.

Sie ist jedoch durch die Vererbung auch kompatibel zu `Person`. Da ein Passagier jedoch kein Pilot ist – ihm fehlt in diesem Beispiel zumindest die Eigenschaft `licenseNumber` –, liefert die letzte Prüfung ein `false`.

## Type Assertion (Type Casting)

Obwohl die im letzten Abschnitt verwendete Variable `person1` auf einen Passagier verweist, erlaubt sie keinen Zugriff auf Eigenschaften, die für Passagiere spezifisch sind, etwa auf den `passengerStatus`:

```
const person1: Person = new Passenger();
person1.firstName = 'Max';
person1.lastName = 'Muster';
// let status = person1.passengerStatus; // FEHLER !!
```

Der Grund dafür ist einfach: Die Variable `person1` ist mit `Person` typisiert. Das bedeutet, dass zur Laufzeit alles, was mit dem Typ `Person` kompatibel ist, zugewiesen werden darf. Das schließt sowohl Passagiere als auch Piloten ein. Auch wenn in diesem einfachen Beispiel auf den ersten Blick ersichtlich ist, dass `person1` nur einen Passagier beherbergt, kann der Compiler das im allgemeinen Fall nicht sicherstellen. Deswegen orientiert er sich bei den erlaubten Zugriffen am Typ der Variablen, also an `Person`. Und eine solche hat eben keinen `passengerStatus`.

Um dieses Problem zu umgehen, muss der Entwickler Verantwortung übernehmen und den Compiler explizit darauf hinweisen, dass es sich um einen Passagier handelt. Hierzu ist in den meisten Sprachen eine Typumwandlung erforderlich. In TypeScript ist hierbei von einer *Type Assertion* die Rede:

```
const person1AsPassenger = person1 as Passenger; // Type Assertion
// let person1AsPassenger = <Passenger>person1;
// alternative Schreibweise
let status = person1AsPassenger.passengerStatus;
```



Das TypeScript-Team nimmt hier bewusst von der in anderen Sprachen üblichen Bezeichnung *Type Cast* Abstand, zumal ein Type Cast häufig auch bestimmte Prüfungen zur Laufzeit involviert. TypeScript hat jedoch keinen Einfluss auf die Ausführung des generierten JavaScript-Codes. Deswegen erscheint auch der Begriff *Type Assertion* treffender, zumal durch ihn ausgedrückt wird, dass man davon ausgeht, zur Laufzeit den gewünschten Typ vorzufinden.

## Abstrakte Klassen

Manche Basisklassen sollen lediglich Felder und Methoden für Subklassen vorgeben, jedoch selbst nicht als Vorlage für Objekte dienen. Ein Beispiel dafür ist die Klasse `AbstractAddress` in Beispiel 2-18. Sie wurde mit dem Schlüsselwort `abstract` versehen.

*Beispiel 2-18: Abstrakte Basisklasse*

```
// src/app/ts/address.ts

export abstract class AbstractAddress {
    id: number = 0;
    street: string = '';
    zipCode: string = '';
    city: string = '';

    constructor(id: number) {
        this.id = id;
    }

    fullAddress(): string {
        return this.street + ', ' + this.zipCode + ' ' + this.city;
    }

    abstract toCSV(): string;
}
```

Das Schlüsselwort `abstract` verhindert nicht, dass andere Klassen von dieser Klasse erben können. Allerdings unterbindet es die Nutzung des `new`-Operators zur Erzeugung einer neuen Instanz von `AbstractAddress`. Die Anweisung

```
const a = new AbstractAddress(7); // FEHLER
```

ist somit nicht zulässig.

Wie dieses Beispiel zeigt, können abstrakte Klassen auch abstrakte Methoden haben. Das sind Methoden, die nur eine Signatur, aber keine Implementierung aufweisen. Demzufolge müssen erbende Klassen sie implementieren (sofern diese nicht auch abstrakt sind). Ein Beispiel dafür ist die `CompanyAddress` (siehe Beispiel 2-19), die von `AbstractAddress` erbt und eine Implementierung für die abstrakte Methode `toCSV` liefert:

*Beispiel 2-19: Erben von abstrakter Basisklasse*

```
// src/app/ts/company-address.ts

import { AbstractAddress } from './address';

export class CompanyAddress extends AbstractAddress {
    companyName: string = '';

    toCSV() {
        return `${this.id};${this.companyName};${this.street};${this.zipCode};
${this.city}`;
    }
}
```

Der Vollständigkeit halber zeigt Beispiel 2-20 den Einsatz der `CompanyAddress`:

*Beispiel 2-20: CompanyAddress instanzieren*

```
// src/app/ts/demo.ts

import { CompanyAddress } from './company-address';

[...]

const a1 = new CompanyAddress(1);
a1.id = 1;
a1.city = 'Graz';
a1.street = 'Hier';
a1.zipCode = '8010';
a1.companyName = 'Steh & Schau GmbH';

console.debug('a1 as csv', a1.toCSV());
console.debug('a1 as full Address', a1.fullAddress());
```

## Zugriff auf die Basisklasse

In manchen Fällen möchte sich eine Subklasse auf die Dienste ihrer Superklasse stützen. Dazu kommt das Schlüsselwort `super` zum Einsatz. Die `PrivateAddress` (siehe Beispiel 2-21), die von der `AbstractAddress` erbt, veranschaulicht die beiden Arten der Nutzung.

*Beispiel 2-21: Zugriff auf die Basisklasse*

```
// src/app/ts/private-address.ts

import { AbstractAddress } from './address';

export class PrivateAddress extends AbstractAddress {
    firstName: string = '';
    lastName: string = '';

    constructor() {
        super();
    }
}
```

```

        fullAddress(): string {
            return this.firstName + ' ' + this.lastName + ', ' + super.fullAddress();
        }

        toCSV(): string {
            return `${this.id};${this.firstName};${this.lastName};${this.street};
${this.zipCode};${this.city}`;
        }
    }
}

```

Zum einen delegiert ihr Konstruktor einen Standardwert an den Konstruktor der Superklasse. Hierzu ruft er `super` auf. Zum anderen überschreibt die Klasse `PrivateAddress` die von der Superklasse bereitgestellte Methode `fullAddress` mit einer eigenen Implementierung und ruft dafür über `super` die Implementierung der Superklasse auf.

Im Gegensatz zu anderen Mainstream-Sprachen vererbt TypeScript Konstruktoren. Wenn eine Subklasse wie die hier betrachtete `PrivateAddress` jedoch einen eigenen Konstruktor aufweist, muss dieser den Konstruktor der Superklasse explizit aufrufen. Genau das ist der Grund dafür, dass der Compiler im betrachteten Fall den Aufruf von `super` erzwingt.

## Ausgewählte Sprachmerkmale

Nachdem wir die wichtigsten Merkmale, die TypeScript für die Entwicklung mit Angular aufweist, anhand eines Beispiels erläutert haben, geht dieser Abschnitt auf ein paar weitere Aspekte ein, die beim Einsatz von Angular nützlich sind.

### Getter und Setter

Nicht alle Felder werden direkt in Objekten gespeichert. Manche müssen auch bei Bedarf berechnet oder von anderen benachbarten Klassen hergeleitet werden. Hierzu stellt TypeScript *Getter* und *Setter* zur Verfügung. Das sind Methoden, die einen Wert ermitteln, etwa durch Berechnen, oder einen Wert aktualisieren. Der Clou daran ist, dass solche Methoden für den Aufrufer wie normale Eigenschaften aussehen. Ein Beispiel dafür bietet die Klasse `ScheduledFlight` in Beispiel 2-22. Sie definiert einen Getter sowie einen Setter `unixDate`, der die Flugzeit als Millisekunden seit dem 1.1.1970 repräsentiert. Diese Information leitet sich aus dem Feld `date` ab.

#### *Beispiel 2-22: Klasse mit Getter und Setter*

```

// src/app/ts/scheduled-flight.ts

import { Flight } from '../flight';

export class ScheduledFlight implements Flight {

    id: number = 0;

```

```

from: string = '';
to: string = '';
date: string = '';

distance: number = 0;

get unixDate(): number {
    return new Date(this.date).getTime();
}

set unixDate(time: number) {
    const date = new Date(time);
    this.date = date.toISOString();
}

calcPrice(): number {
    return this.distance / 3;
}

}

```

Achten Sie darauf, dass nach den Schlüsselwörtern `get` und `set` ein Leerzeichen steht. Im Gegensatz zu Java ist `get` bzw. `set` also nicht Teil des Namens, sondern zeigt nur an, dass jetzt eine Methode folgt, die als Getter bzw. Setter fungieren soll.

Für den Aufrufer gibt es keinen erkennbaren Unterschied zwischen herkömmlichen Feldern und solchen, die über Getter und Setter definiert werden: Er setzt einfach die Eigenschaft `unixDate` oder liest sie aus (siehe Beispiel 2-23).

#### *Beispiel 2-23: ScheduledFlight instanzieren*

```

// src/app/ts/demo.ts
import { ScheduledFlight } from './scheduled-flight';

[...]

const nextFlight = new ScheduledFlight();
nextFlight.date = '2018-12-24';
console.debug('unix-date', nextFlight.unixDate);
nextFlight.unixDate = 1000;
console.debug('unix-date', nextFlight.date);

```

## Generics

Nicht immer lässt sich im Voraus ein Datentyp festlegen. *Generische Klassen*, kurz *Generics*, helfen in solchen Situationen. Sie erhalten Typparameter für jene Datentypen, die beim Schreiben der Klasse nicht bekannt sind. Diese Typparameter können Sie verwenden wie andere Datentypen auch. Erst wenn die Klasse instanziert wird, ist für diese Typparameter ein konkreter Datentyp anzugeben.

Bei der Klasse `Invoice` in Beispiel 2-24 handelt es sich um solch ein Generic.

*Beispiel 2-24: Generische Klasse*

```
// src/app/ts/invoice.ts

export class Invoice<T> {

    constructor(readonly subject: T, readonly price: number) {}

    toString(): string {
        let id = '';
        // id = this.subject.id; // Fehler
        return `${id}: ${this.price}`;
    }
}
```

Sie definiert einen Typparameter `T` im Anschluss an den Klassennamen und nutzt diesen zur Typisierung der Eigenschaft `subject`.

Ein Problem gibt es hier jedoch in der Methode `toString`: Der Compiler verbietet den Zugriff auf `this.subject.id`. Der Typparameter `T` kann zur Laufzeit ja alles Mögliche sein, und deswegen kann der Compiler nicht garantieren, dass dann eine Eigenschaft `id` vorliegt.

Abhilfe schafft hier eine Einschränkung für den Typparameter. Die Klasse `FlightInvoice` in Beispiel 2-25 gibt beispielsweise mit `extends Flight` an, dass der Typparameter `T` ein `Flight` oder eine Subklasse davon sein muss:

*Beispiel 2-25: Generische Klasse mit Einschränkung für Typparameter*

```
// src/app/ts/flight-invoice.ts

import { Flight } from '../flight';

export class FlightInvoice<T extends Flight> {

    constructor(readonly subject: T, readonly price: number) {}

    toString(): string {
        const id = this.subject.id; // Klappt nun!
        return `${id}: ${this.price}`;
    }
}
```

Aufgrund dieser Einschränkung garantiert TypeScript Zugriff auf alle Felder, die ein `Flight` mit sich bringt. Dazu gehört auch die `id`.

Beim Instanziieren eines Generic ist der gewünschte Datentyp anzugeben. Ein Beispiel dafür findet sich in Beispiel 2-26, das `T` auf `CharterFlight` festlegt:

### Beispiel 2-26: Nutzung einer generischen Klasse

```
// src/app/ts/demo.ts
import { CharterFlight } from './charter-flight';
import { FlightInvoice } from './flight-invoice';

[...]

const charterFlightToCharge = new CharterFlight();
charterFlightToCharge.from = 'Graz';
charterFlightToCharge.to = 'Hamburg';
charterFlightToCharge.distance = 1000;

const price = charterFlightToCharge.calcPrice();
const charterInvoice = new FlightInvoice<CharterFlight>(charterFlightToCharge, price);

console.debug('charterInvoice', charterInvoice.toString());
```

Neben Typen wie Klassen lassen sich auch Funktionen sowie Methoden generisch gestalten:

```
function min<T>(a: T, b: T): T {
    if (a < b) {
        return a;
    }
    return b;
}
```

In diesem Fall ist gewährleistet, dass die beiden Übergabeparameter sowie der Rückgabewert vom selben Typ sind. Beim Aufruf können Sie nun einen konkreten Datentyp für den festgelegten Typparameter T angeben:

```
const r1 = min<number>(1, 2); // 1
```

In vielen Fällen kann TypeScript den Datentyp des Typparameter jedoch aus dem Aufruf ableiten. Somit können Sie häufig die Angabe des Datentyps weglassen:

```
const r2 = min(1, 2); // 1
const r3 = min('a', 'b'); // 'a'
```

## Exceptions

Zur Fehlerbehandlung unterstützen JavaScript und TypeScript *Ausnahmen*, wie auch viele andere Sprachen. Im Englischen heißen Ausnahmen *Exceptions*. Eine Funktion oder Methode löst eine Exception aus, um den Aufrufer auf einen Fehlerzustand hinzuweisen. Dieses Auslösen wird auch als *Werfen* bezeichnet. Ansonsten liefert sie das gewünschte Ergebnis.

Um eine Exception zu werfen, kommt das Schlüsselwort `throw` zum Einsatz (siehe Beispiel 2-27):

*Beispiel 2-27: Eine Exception werfen*

```
// src/app/ts/demo.ts  
[...]  
  
function div(a: number, b: number): number {  
    if (b === 0) {  
        throw new Error('division by 0 is not allowed');  
    }  
    return a / b;  
}
```

Im Gegensatz zu vielen anderen Sprachen kann `throw` jeden beliebigen Wert werfen, sogar eine `number` oder einen `string`. Allerdings ist es üblich, dass eine Instanz von `Error` oder einer Subklasse von `Error` zum Einsatz kommt, um den jeweiligen Fehler anzuzeigen.

Eine Funktion oder Methode, die eine Exception wirft, kann in einem `try`-Block aufgerufen werden (siehe Beispiel 2-28):

*Beispiel 2-28: Eine Exception fangen*

```
// src/app/ts/demo.ts  
[...]  
  
try {  
    let result1 = div(10, 3);  
    console.debug('result1', result1);  
  
    let result2 = div(10, 0);  
    console.debug('result2', result2);  
}  
catch (error) {  
    console.error('Fehler', error);  
}  
finally {  
    console.debug('finally');  
}
```

Sobald eine Exception auftritt, bricht JavaScript den `try`-Block ab und springt in den nächsten `catch`-Block, der das geworfene Element entgegennimmt. Für Aufräumarbeiten kann der `try`-Block auch einen `finally`-Block erhalten. Dieser wird in jedem Fall ausgeführt – egal ob ein Fehler aufgetreten ist oder nicht.

Ein `try`-Block muss immer mit einem `catch`- oder einem `finally`-Block auftreten. Auch alle drei Blöcke können kombiniert werden, wie das aktuelle Beispiel veranschaulicht.

## Spread-Operator

Es gibt immer wieder Fälle, in denen man ein bestehendes Objekt nicht verändern darf, weil zum Beispiel auch noch andere Programmteile davon abhängig sind. In diesen Fällen bietet es sich an, das Objekt für die eigenen Zwecke zu klonen. Ein sehr nützliches Sprachmerkmal hierfür ist der Spread-Operator:

```
const flight: Flight = {  
    id: 1,  
    from: 'Graz',  
    to: 'Hamburg',  
    date: '2018-12-24T17:00:00.0001:00'  
}  
  
const flightClone1 = { ...flight };  
const flightClone2 = { ...flight, id: 2 };
```

Während `flightClone1` eine Kopie von `flight` erhält, bekommt `flightClone2` eine modifizierte Kopie, die eine andere ID aufweist.

Dasselbe funktioniert auch mit Arrays:

```
const ary = [1, 2, 3, 4];  
const aryClone1 = [...ary];  
const aryClone2 = [0, ...ary];
```

Hier ist `aryClone1` ebenfalls eine Kopie und `aryClone2` eine Kopie, der zusätzlich ein Eintrag 0 hinzugefügt wurde.

Wichtig zu wissen ist, dass der Spread-Operator immer »nur« eine flache Kopie erzeugt, d.h., untergeordnete Objekte kopiert er nicht mit. Wollen Sie diese ebenfalls kopieren, müssen Sie sich selbst darum kümmern:

```
const obj = {  
    id: 1,  
    // untergeordnetes Objekt:  
    more: {  
        count: 10  
    }  
};  
  
const objClone = {...obj, more: {...obj.more}}
```

## Strikte Null-Prüfungen

So gut wie jede Sprache bietet Nullwerte, um anzudeuten, dass für eine Variable kein Wert vorliegt. JavaScript und TypeScript bieten daneben sogar noch den Wert `undefined`. Dieser informiert darüber, dass die gewünschte Eigenschaft gar nicht vorhanden ist. Versucht der Programmcode, eine Eigenschaft oder Methode bei einer Variablen mit dem Wert `null` oder `undefined` zu nutzen, ergibt sich ein Laufzeitfehler. Da standardmäßig jede Variable diese Werte aufnehmen kann, müsste streng genommen der Programmcode mit Prüfungen sicherstellen, dass der gewünschte Zugriff sicher ist. Beispiel 2-29 veranschaulicht das anhand des in diesem Kapitel verwendeten `FlightManager`:

### Beispiel 2-29: Auf Nullwerte prüfen

```
// src/app/ts/flight-manager.ts
import { Flight } from '../flight';

export class FlightManager {

    private cache: Flight[];

    constructor(cache: Flight[]) {
        if (!cache) throw Error('null or undefined is not allowed!');
        this.cache = cache;
    }

    [...]
}
```

Hier prüft der Konstruktor den übergebenen cache auf null bzw. undefined. Streng genommen, prüft er, ob der Wert *falsy* ist, was null und undefined inkludiert (siehe den Abschnitt »boolean« auf Seite 42). Damit verhindert das Beispiel, dass die Methode search beim Iterieren des Caches auf einen Fehler läuft, wenn der Cache einen Nullwert aufweist.

Leider führt diese Prüfung nur dazu, dass die falsche Verwendung des FlightManager etwas früher zur Laufzeit entdeckt wird. Wünschenswert wäre es jedoch, solche Fehler schon beim Kompilieren ausschließen zu können. Hierzu bietet TypeScript strikte Null-Prüfungen (*Strict Null Checks*).

Da die Angular CLI standardmäßig den Strict Mode aktiviert, kommen wir auch automatisch in den Genuss der Strict Null Checks. Um sie zu deaktivieren, könnten Sie die Eigenschaft `strictNullChecks` in der `tsconfig.json` auf `false` setzen:

```
{
    'compilerOptions': {
        [...]
        'strictNullChecks': true
    }
}
```

Wir wollen es hier jedoch damit belassen. Bei Nutzung dieser Option können typisierte Variablen standardmäßig weder den Wert `null` noch `undefined` aufnehmen. Um dennoch diese Werte zuweisen zu können, muss die Deklaration das explizit erlauben. Der Aufruf

```
let fm = new FlightManager(null);
```

provoziert somit einen Compilerfehler. Um wieder Nullwerte zuzulassen, sind sogenannte *Union-Types* zu nutzen. Dabei handelt es sich um ein Sprachkonstrukt von TypeScript, das vorgibt, dass eine Variable einen von mehreren Typen aufweisen muss. Die Zeile

```
let stringOrNumber: string | number;
```

deklariert beispielsweise eine Variable, die sowohl einen String als auch eine Zahl aufnehmen kann.

Auf die gleiche Weise lassen sich die Werte `null` und `undefined` in den Wertebereich einer Variablen aufnehmen. Allerdings sind in diesem Fall wieder die eingangs erwähnten Prüfungen nötig. Beispiel 2-30 veranschaulicht das und kompensiert das Fehlen des Caches mit einem leeren Array:

*Beispiel 2-30: Die Werte `null` und `undefined` in den Wertebereich aufnehmen*

```
// src/app/ts/flight-manager.ts
import { Flight } from '../flight';

export class FlightManager {

    private cache: Flight[];

    constructor(cache: Flight[] | null | undefined) {
        if (!cache) {
            cache = [];
        }
        this.cache = cache;
    }

    [...]
}
```

In diesem Fall würde der Compiler ohne Null-Prüfung sogar einen Fehler auslösen, da `this.cache` weder `null` noch `undefined` aufnehmen kann.

## Asynchrone Operationen

Zum Abschluss beschäftigt sich dieses Kapitel mit einem etwas komplizierteren Thema, um das man in Browseranwendungen kaum herumkommt: Asynchronität. Um zu verhindern, dass das Browserfenster einfriert, führt der Browser alles, was etwas länger dauern kann, im Hintergrund aus – also asynchron. Sobald die gewünschte Aktion abgeschlossen ist, ruft er Callback-Funktionen im Hauptthread auf, der sich um das Rendering im Browserfenster kümmert.

Das Paradebeispiel dafür ist der Zugriff auf Web-APIs über das Browserfenster. Hier wäre es nämlich schade, wenn das Browserfenster einfrieren würde, nur weil der Browser auf Daten vom Server wartet. Genau das Thema greift dieser Abschnitt auf, um den Umgang mit asynchronen Operationen zu veranschaulichen.

Als Beispiel für asynchrone Operationen nutzen wir zur Vereinfachung Time-outs. Sehen Sie dies bitte stellvertretend für alle anderen möglichen asynchronen Operationen, wie z.B. das Abrufen von Daten via HTTP. Letzteres wird in den folgenden Kapiteln genauer beschrieben und immer wieder verwendet.

## Callbacks und die Pyramide of Doom

Anders als klassische Operationen liefern asynchrone Operationen nicht unmittelbar ein Ergebnis. Stattdessen stoßen sie häufig eine registrierte Funktion – einen sogenannten Callback – an, sobald das asynchron ermittelte Ergebnis vorliegt:

```
setTimeout(() => {
    console.log('Timeout!');
}, 1000);
```

In diesem Fall wird der Callback in Form eines Lambda-Ausdrucks übergeben. Der zweite Parameter gibt an, nach welcher Zeitspanne `setTimeout` den Callback ausführen soll. Hier wird dieser Wert auf 1000 Millisekunden festgelegt.

Leider sind asynchrone Operationen schwer zu lesen, zumal die JavaScript-Engine sie nicht sequenziell ausführt. Im Fall von

```
setTimeout(() => {
    console.log('Timeout!');
}, 1000);

console.log('Start!');
```

ergibt sich zum Beispiel die Ausgabe `Start!` und `Timeout!`, obwohl sich die Reihenfolge der beiden Ausgaben im Quellcode genau andersherum gestaltet.

Noch unleserlicher wird es, wenn Callbacks weitere asynchrone Operationen anstoßen:

```
setTimeout(() => {
    console.log('Phase 1');
    setTimeout(() => {
        console.log('Phase 2');
        setTimeout(() => {
            console.log('Phase 3');
            setTimeout(() => {
                console.log('Phase 4');
            }, 1000);
        }, 1000);
    }, 1000);
}, 1000);
```

Hier ergeben sich nach jeweils 1.000 Millisekunden die Ausgaben Phase 1, Phase 2, Phase 3 und Phase 4.

Bei Codestrecken wie dieser ist auch von der *Pyramide of Doom* die Rede – zu Deutsch etwa »Pyramide der Verdammnis«. Diese Bezeichnung bezieht sich auf die Form der Einrückungen auf der linken Seite.

Um Quellcode dieser Art zu vermeiden, wurden die sogenannten Promises entwickelt. Der nächste Abschnitt geht auf sie ein.

## Promises

Promises sind Datenstrukturen, die mittlerweile Teil des ECMAScript-Standards sind. Sie repräsentieren Werte, die in der Regel asynchron ermittelt werden und bereits vorliegen können, aber nicht vorliegen müssen.

Der Aufrufer registriert bei einem Promise einen oder zwei Callbacks. Der eine nimmt das Ergebnis entgegen, sobald es vorliegt. Den anderen stößt der Promise im Fehlerfall an.

Um das zu veranschaulichen, erzeugt die nachfolgende Funktion einen Promise, der die oben verwendete Funktion `setTimeout` kapselt:

```
function timeout(time: number): Promise<number> {
    return new Promise((resolve, reject) => {
        if (time < 0) {
            // Send error!
            reject(`Don't be that negative!`);
            return;
        }

        setTimeout(() => {
            resolve(time);
        }, time);
    });
}
```

Da es sich hier um einen `Promise<number>` handelt, kann der Aufrufer davon ausgehen, dass es sich beim gelieferten Wert um eine `number` handelt.

Der instanzierte Promise bekommt zwei Funktionen übergeben: Die erste erhält in der Regel den Namen `resolve`, die zweite normalerweise `reject`. Sobald die asynchrone Operation erfolgreich ausgeführt wurde, ist `resolve` aufzurufen und dabei das Ergebnis zu übergeben. Im Fehlerfall kommt `reject` zur Ausführung. Diese Funktion nimmt eine Fehlermeldung entgegen.

Unsere Hilfsmethode `timeout` lässt sich nun wie folgt aufrufen:

```
timeout(1000)
    .then(result => console.debug('Timeout!', result))
    .catch(error => console.debug('error', error));
```

Der mit `then` registrierte Callback nennt sich Fulfillment-Handler. Er erhält das ermittelte Ergebnis, sobald es vorliegt. Der mit `catch` registrierte Callback ist der sogenannte Rejection-Handler. Der Promise bringt ihn im Fehlerfall zur Ausführung.

Dieser Aufruf sieht nicht viel anders aus als der im letzten Abschnitt. Tatsächlich entfalten Promises erst dann ihr Potenzial, wenn der Aufrufer sie verkettet:

```
timeout(1000)
    .then(() => {
        console.debug('Phase 1!', result);
        return timeout(1000);
```

```

})
.then(() => {
    console.debug('Phase 2!', result);
    return timeout(1000);
})
.then(() => {
    console.debug('Phase 3!', result);
    return timeout(1000);
})
.then(() => {
    console.debug('Phase 4!', result);
    return timeout(1000);
})
.catch(error => console.error('error', error));

```

Liefert ein Handler einen weiteren Promise, erhält der nächste Fulfillment-Handler das Ergebnis, sobald es vorliegt. Im Fehlerfall ruft der Promise den nächsten Rejection-Handler auf.

Das ist bereits eine erste Verbesserung gegenüber der weiter oben gezeigten Pyramide of Doom. Es geht aber noch ein wenig besser, wie der nächste Abschnitt zeigt.

## async und await

Seit ECMAScript 2017 können Sie Funktionen, die einen Promise liefern, mit dem Schlüsselwort `async` markieren:

```

async function timeout2(time: number): Promise<number> {
    return new Promise((resolve, reject) => {

        if (time < 0) {
            // Send error!
            reject(`Don't be that negative!`);
            return;
        }

        setTimeout(() => {
            resolve(time);
        }, time);
    });
}

```

Interessant wird es erst, wenn Sie solche Funktionen mit dem Schlüsselwort `await` anstoßen:

```

async function caller() {
    let result;

    try {
        result = await timeout2(1000);
        console.log('Phase 1', result);

        result = await timeout2(1000);
        console.log('Phase 2', result);
    }
}

```

```

        result = await timeout2(1000);
        console.log('Phase 3', result);

        result = await timeout2(1000);
        console.log('Phase 4', result);
    }
    catch (error) {
        console.error('error', error);
    }
}

caller();

```

Die Zeilen nach dem `await`-Aufruf registriert ECMAScript als Fulfillment-Handler. `catch`-Blöcke werden hingegen ohne weiteres Zutun zum Rejection-Handler. Dank dieser Schlüsselwörter können wir den Quellcode nun sequenziell und ohne zusätzliche `then`- und `catch`-Aufrufe gestalten.



Bitte beachten Sie, dass das Schlüsselwort `await` nur innerhalb von mit `async` gekennzeichneten Operationen verwendet werden darf.

## Bedeutung von Promises in Angular

Promises gehören auf jeden Fall zum Handwerkszeug jedes JavaScript-Entwicklers und jeder JavaScript-Entwicklerin, unter anderem weil sie mittlerweile Teil des ECMAScript-Standards sind und allein schon deswegen häufiger vorkommen. Auch wir werden in diesem Buch immer wieder auf Stellen stoßen, an denen Angular Promises unterstützt. Häufiger als Promises nutzt Angular jedoch sogenannte *Observables*. Dabei handelt es sich um ein auf den ersten Blick ähnliches Konzept, das jedoch um einiges mächtiger ist. Den ersten Kontakt mit Observables werden wir bereits in Kapitel 3 haben, wenn wir Daten via HTTP abrufen.

Daneben sind Promises sowie die Schlüsselwörter `async` und `await` bei der Automatisierung von Testfällen sehr nützlich. Kapitel 12 bietet dazu einige Details.

## Zusammenfassung

Die Sprache TypeScript ist sehr nahe am offiziellen JavaScript-Standard und bietet zusätzlich ein statisches Typsystem, das beim frühzeitigen Erkennen von Fehlern hilft. In vielen Fällen kann TypeScript sogar den korrekten Datentyp herleiten, ohne dass Sie ihn explizit angeben müssen.

Daneben bietet TypeScript einige Konstrukte, die sich auch in anderen Sprachen bewährt haben, beispielsweise Generics, Interfaces oder Lambda-Ausdrücke. Um überall lauffähig zu sein, lassen sich TypeScript-Programme in verschiedene ECMAScript-Versionen übersetzen.

# Eine erste Angular-Anwendung

Um Ihnen die einzelnen Aspekte von Angular zu vermitteln, verwenden wir in diesem Buch ein durchgängiges Beispiel. Sie können es unter <https://www.ANGULARArchitects.io/leser> herunterladen. Dabei handelt es sich um eine Anwendung zum Buchen von Flügen.

In diesem Abschnitt entwickeln wir für die in Kapitel 1 generierte Anwendung einen ersten Teil davon, um Ihnen die Grundlagen von Angular zu vermitteln. Dieser Anwendungsteil kümmert sich um das Suchen von Flugverbindungen. Außerdem können Sie den gewünschten Flug auswählen. Abbildung 3-1 gibt einen Vorgeschnack darauf.

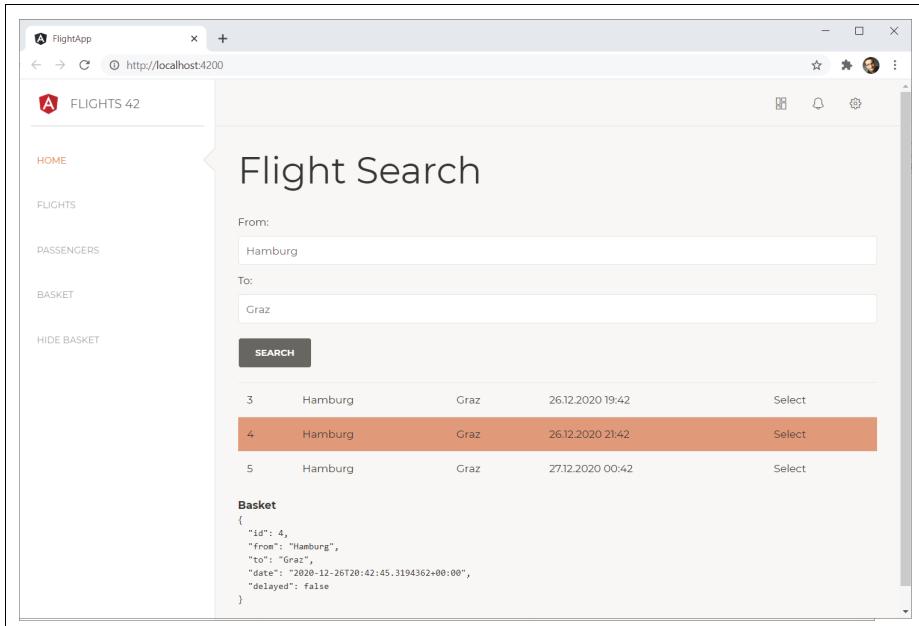


Abbildung 3-1: Anwendung zum Suchen nach Flügen



Um die hier beschriebenen Ausführungen nachzustellen, benötigen Sie eine aktuelle Version von NodeJS sowie die Angular CLI. Darüber hinaus bietet sich der Einsatz eines Editors mit Unterstützung für TypeScript an, z.B. Visual Studio Code. Außerdem gehen wir davon aus, dass Sie mit der Angular CLI bereits eine neue Angular-Anwendung generiert haben (`ng new`). Informationen zu beiden Themen finden Sie in Kapitel 1.

## Angular-Komponente erzeugen

Als Erstes werden wir eine Angular-Komponente für den besprochenen Anwendungsfall erstellen. Wechseln Sie dazu auf die Konsole. Führen Sie im Hauptverzeichnis der Anwendung den folgenden Befehl aus:

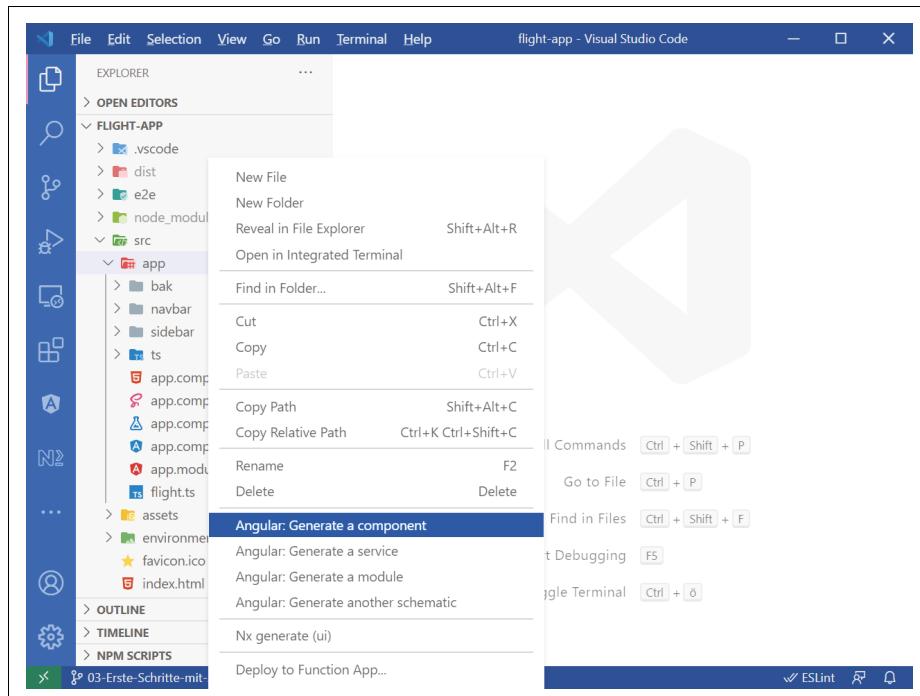
```
ng generate component flight-search
```



Die Befehle der CLI lassen sich abkürzen, die betrachtete Anweisung könnte man beispielsweise auch wie folgt formulieren:

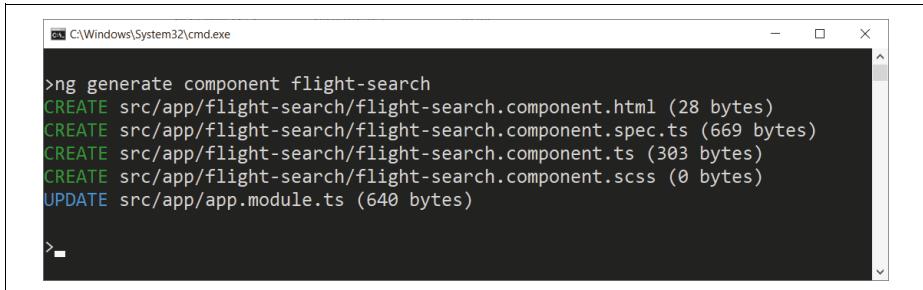
```
ng g c flight-search
```

Mit dem in Kapitel 1 erwähnten Plug-in *Angular Schematics* lässt sich dieser CLI-Befehl auch direkt über Visual Studio Code anstoßen. Wählen Sie dazu die Anweisung *Angular: Generate a component* aus dem Kontextmenü des gewünschten Ordners.



Nach dem Auswählen dieser Anweisung stellt Ihnen Visual Studio Code mehrere Fragen. Die Frage nach dem Komponentennamen beantworten Sie analog zum oben diskutierten Befehl mit `flight-search`. Die anderen Fragen können Sie einfach mit *Enter* quittieren, um mit den Standardeinstellungen der CLI vorlieb zu nehmen.

Die Angular CLI generiert daraufhin mehrere Dateien für die gewünschte Komponente (siehe Abbildung 3-2).



```
C:\Windows\System32\cmd.exe
>ng generate component flight-search
CREATE src/app/flight-search/flight-search.component.html (28 bytes)
CREATE src/app/flight-search/flight-search.component.spec.ts (669 bytes)
CREATE src/app/flight-search/flight-search.component.ts (303 bytes)
CREATE src/app/flight-search/flight-search.component.scss (0 bytes)
UPDATE src/app/app.module.ts (640 bytes)

>-
```

Abbildung 3-2: Komponente zum Suchen nach Flügen mit der CLI generieren

Alle diese Dateien richtet die CLI im Ordner `src/app/flight-search` ein:

#### `flight-search.component.html`

Das Template der Komponente. Es bestimmt, wie Angular die Komponente darstellt.

#### `flight-search.component.spec.ts`

Ein Unit-Test für die Komponente. Details zum Thema Testen finden Sie in Kapitel 12.

#### `flight-search.component.ts`

Die TypeScript-Klasse, die die Komponente repräsentiert. Sie definiert das gewünschte Verhalten.

#### `flight-search.component.scss`

Die Stylesheet-Datei mit lokalen Styles für unsere Komponente.

Die Dateien `flight-search.component.ts` und `flight-search.component.html` werden wir in den nachfolgenden Abschnitten näher betrachten und für unsere Zwecke anpassen.

## Komponentenlogik

Die generierte Datei `flight-search.component.ts` beinhaltet das Grundgerüst für unsere Komponentenlogik (siehe Beispiel 3-1).

#### Beispiel 3-1: Generierte Komponente

```
// src/app/flight-search/flight-search.component.ts
import { Component, OnInit } from '@angular/core';
```

```

@Component({
  selector: 'app-flight-search',
  templateUrl: './flight-search.component.html',
  styleUrls: ['./flight-search.component.scss']
})
export class FlightSearchComponent implements OnInit {

  constructor() { }

  ngOnInit(): void {
  }
}

```

Viele der hier generierten Konstrukte haben wir bereits in Kapitel 1 im Rahmen der AppComponent besprochen. Allerdings möchten wir hier Ihre Aufmerksamkeit auf ein paar Details lenken:

- Der Selektor lautet `app-flight-search`. Das Präfix `app` wurde von der CLI eingefügt. Diese Präfixe sollen Namenskonflikte mit Komponenten aus Bibliotheken verhindern.
- Die generierte Klasse nennt sich `FlightSearchComponent`, während die zugrunde liegende Datei den Namen `flight-search.component.ts` erhalten hat. Hierbei handelt es sich um die üblichen Namenskonventionen in der Welt von Angular.
- `FlightSearchComponent` implementiert das Interface `OnInit`, das wiederum die Methode `ngOnInit` vorgibt. Diese Methode ruft Angular nach dem Initialisieren der Komponente auf, und somit kann sie für Initialisierungen von Eigenschaften verwendet werden. Details dazu finden Sie in Kapitel 4.

Lassen Sie uns nun dieses Grundgerüst ein wenig ausbauen, um eine Suche nach Flügen zu ermöglichen (siehe Beispiel 3-2).

#### *Beispiel 3-2: Eigenschaften und Methoden für die Flugsuche*

```

// src/app/flight-search/flight-search.component.ts

import { Component, OnInit } from '@angular/core';
import { Flight } from '../flight';

@Component({
  selector: 'app-flight-search',
  templateUrl: './flight-search.component.html',
  styleUrls: ['./flight-search.component.scss']
})
export class FlightSearchComponent implements OnInit {

  from = 'Hamburg';
  to = 'Graz';
  flights: Array<Flight> = [];
  selectedFlight: Flight | null = null;

  constructor() {
  }
}

```

```

ngOnInit(): void {
}

search(): void {
    // Implementierung folgt weiter unten.
}

select(f: Flight): void {
    this.selectedFlight = f;
}

}

```

Die Eigenschaften `from` und `to` repräsentieren die Suchkriterien für die gewünschten Flüge. Die Standardwerte sollen hier verhindern, dass wir später immer wieder die gleichen Suchkriterien eingeben müssen. Außerdem lassen sie uns auf den ersten Blick erkennen, ob der weiter unten angestrebte automatische Abgleich zwischen den Eigenschaften und den Textfeldern funktioniert.

Das Array `flights` nimmt die gefundenen Flüge auf. Es ist mit dem Interface `Flight` aus Kapitel 1 typisiert. Falls Sie dieses Kapitel übersprungen haben, finden Sie den hier benötigten Teil dieses Interface in Beispiel 3-3. Ein paar der in Kapitel 1 verwendeten optionalen Eigenschaften wurden zur Vereinfachung weggelassen.

Die Eigenschaft `selectedFlight` repräsentiert den ausgewählten Flug. Damit sie initial den Wert `null` bekommen kann, ist sie vom Typ `Flight | null` (mehr Informationen über strikte Null-Prüfungen gibt es in Kapitel 2).

Die Methode `search` kümmert sich um das Abrufen der Flüge, und `select` notiert sich den vom Benutzer ausgewählten Flug.

#### *Beispiel 3-3: Interface für einen Flug*

```

// src/app/flight.ts

export interface Flight {
    id: number;
    from: string;
    to: string;
    date: string; // ISO-Datum: 2016-12-24T17:00+01:00
    delayed?: boolean;
}

```

## Auf das Backend zugreifen

Für Ihre Hauptaufgabe muss die `FlightSearchComponent` via HTTP auf eine Web-API mit Flügen zugreifen. Für solche Vorhaben bietet Angular die Klasse `HttpClient`. Da diese Klasse wiederverwendbare Dienste anbietet, ist hierbei auch von einem Service die Rede.

Um Zugriff auf den Service zu bekommen, müssen Sie zunächst das `HttpClient` Module in Ihr `AppModule` importieren (siehe Beispiel 3-4).

*Beispiel 3-4: HttpClientModule in das AppModule importieren*

```
// src/app/app.module.ts

[...]
// Diese Zeile einfügen:
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  imports: [
    //Diese Zeile unter *imports* einfügen:
    HttpClientModule,
    BrowserModule
  ],
  declarations: [
    [...]
  ],
  providers: [],
  bootstrap: [
    AppComponent
  ]
})
export class AppModule { }
```

Danach können Sie über den Konstruktor der FlightSearchComponent eine Instanz von HttpClient anfordern (siehe Beispiel 3-5).

*Beispiel 3-5: HttpClient über Konstruktorargument anfordern*

```
// src/app/flight-search/flight-search.component.ts

import { HttpClient } from '@angular/common/http';
import { Component, OnInit } from '@angular/core';
import { Flight } from '../flight';

@Component({
  selector: 'app-flight-search',
  templateUrl: './flight-search.component.html',
  styleUrls: ['./flight-search.component.scss']
})
export class FlightSearchComponent implements OnInit {

  from = 'Hamburg';
  to = 'Graz';
  flights: Array<Flight> = [];
  selectedFlight: Flight | null = null;

  // HttpClient anfordern.
  constructor(private http: HttpClient) {
  }

  [...]
}
```

Diese Vorgehensweise nennt sich auch *Dependency Injection* bzw. *Constructor Injection*: Die benötigte Serviceinstanz wird demnach von Angular in den Konstruktor injiziert. Das bedeutet, dass Angular entscheidet, welche konkrete Ausprägung des HttpClient die Komponente erhält. Während Angular für den Produktivbetrieb den »richtigen« HttpClient erzeugt, könnte es für automatisierte Tests eine Dummy-Implementierung verwenden, die HTTP-Zugriffe lediglich simuliert.

Weitere Details zu diesem Mechanismus finden sich in Kapitel 12.

Da wir nun unsere HttpClient-Instanz haben, können wir damit innerhalb von search auf die Web-API zugreifen (siehe Beispiel 3-6).

*Beispiel 3-6: Implementierung von search*

```
// src/app/flight-search/flight-search.component.ts

// Wir benötigen diese drei Importe für den HttpClient:
import { HttpClient, HttpHeaders, HttpParams } from '@angular/common/http';

import { Component, OnInit } from '@angular/core';
import { Flight } from './flight';

@Component({
  selector: 'app-flight-search',
  templateUrl: './flight-search.component.html',
  styleUrls: ['./flight-search.component.scss']
})
export class FlightSearchComponent implements OnInit {

  from = 'Hamburg';
  to = 'Graz';
  flights: Array<Flight> = [];
  selectedFlight: Flight | null = null;

  constructor(private http: HttpClient) {}

  ngOnInit(): void {}

  search(): void {

    const url = 'http://demo.ANGLARarchitects.io/api/flight';

    const headers = new HttpHeaders()
      .set('Accept', 'application/json');

    const params = new HttpParams()
      .set('from', this.from)
      .set('to', this.to);

    this.http.get<Flight[]>(url, {headers, params}).subscribe({
      next: (flights) => {
        this.flights = flights;
      },
    });
  }
}
```

```

        error: (err) => {
          console.error('Error', err);
        });
      });
    }

    select(f: Flight): void {
      this.selectedFlight = f;
    }
  }
}

```

Die Methode `search` ruft nun via HTTP Flüge ab und hinterlegt sie in der Eigenschaft `flights`:

- Die zu nutzenden HTTP-Kopfzeilen stellt der `HttpClient` mit einer Instanz von `HttpHeaders` dar. Das Beispiel übergibt die Kopfzeile `Accept`, um anzugeben, dass wir JSON als Antwortformat wünschen. Dabei handelt es sich um das einzige Datenformat, das Angular ab Werk unterstützt.
- Die zu übersendenden URL-Parameter repräsentiert der `HttpClient` mit einer `HttpParams`-Auflistung.
- Bitte beachten Sie, dass die beiden Aufrufe von `set` die aktuelle Auflistung *nicht verändern*, sondern eine neue Auflistung zurückliefern. Deswegen verkettet das Beispiel auch die einzelnen Aufrufe von `set`.
- Die Methode `get` führt einen HTTP-Zugriff unter Verwendung der HTTP-Methode `GET` durch. Diese Methode kommt typischerweise zum Abrufen von Daten zum Einsatz.
- Als Ergebnis des HTTP-Aufrufs erwartet der `HttpClient` ein JSON-Dokument, das er in ein JavaScript-Objekt umwandelt. Den Datentyp dieses Objekts nimmt `get` als Typparameter entgegen (`Flight[]`).
- Das Abrufen von Daten erfolgt im Browser asynchron, also im Hintergrund. Sobald die Daten vorliegen, bringt der `HttpClient` eine der beiden bei `subscribe` registrierten Methoden zur Ausführung: `next` im Erfolgsfall und `error` in Fehlerfall. Das Objekt, das die Methode `subscribe` anbietet, ist übrigens ein sogenanntes *Observable*. Mehr zu diesem Thema findet sich in Kapitel 11.

Neben der hier verwendeten Methode `get` bietet der `HttpClient` noch weitere Methoden für andere Arten von HTTP-Zugriffen.

Tabelle 3-1: Methoden von `HttpClient`

Methode	Semantik
<code>get&lt;T&gt;(url, options)</code>	Abrufen von Ressourcen.
<code>post&lt;T&gt;(url, body, options)</code>	Hinzufügen einer Ressource oder Anstoßen einer Verarbeitung am Server.
<code>put&lt;T&gt;(url, body, options)</code>	Hinzufügen oder Aktualisieren einer Ressource.

Tabelle 3-1: Methoden von HttpClient (Fortsetzung)

Methode	Semantik
patch<T>(url, body, options)	Aktualisieren einer Ressource. Es müssen nur die geänderten Eigenschaften übergeben werden.
delete<T>(url, options)	Löschen einer Ressource.

Der Begriff *Ressource* kommt aus der Welt von HTTP und bezeichnet das abgerufene oder zu sendende Objekt bzw. Dokument. Der Typparameter T steht für den Datentyp der Antwort. Im oben betrachteten Beispiel war das Flight[ ]. Jene Methoden, die Daten zum Server senden, weisen einen Parameter body auf. Dieser nimmt das zu sendende Objekt entgegen. Für die Übertragung per HTTP wandelt der HttpClient es in ein JSON-Objekt um. Der Parameter options erhält ein Objekt, das die HTTP-Anfrage näher beschreibt. Im oben gezeigten Beispiel verweist es auf die zu sendenden Kopfzeilen sowie auf die zu verwendenden URL-Parameter.

Bitte beachten Sie auch, dass nicht jede Web-API alle hier beschriebenen Methoden unterstützt. Außerdem muss die implementierte Semantik dieser Methoden nicht jener von HTTP beschriebenen entsprechen. Beispielsweise könnten sich die Autoren einer Web-API entscheiden, aufgrund eines empfangenen post-Aufrufs serverseitige Ressourcen zu aktualisieren, obwohl hierfür put oder patch vorgesehen ist. Aufschluss darüber bietet jeweils die Dokumentation der einzubindenden Web-API.

Zur Veranschaulichung erzeugt die Methode in Beispiel 3-7 einen neuen Flug.

*Beispiel 3-7: Einen neuen Flug mit post erzeugen*

```
createDemoFlight(): void {
  const url = 'http://demo.ANGLARarchitects.io/api/flight';

  const headers = new HttpHeaders()
    .set('Accept', 'application/json');

  const newFlight: Flight = {
    id: 0,
    from: 'Gleisdorf',
    to: 'Graz',
    date: new Date().toISOString()
  };

  this.http.post<Flight>(url, newFlight, { headers }).subscribe({
    next: (flight) => {
      console.debug('Neue Id: ', flight.id);
    },
    error: (err) => {
      console.error('Error', err);
    }
  });
}
```

Das Beispiel geht davon aus, dass der erzeugte Flug samt der serverseitig vergebenen ID wieder zurückgeliefert wird.

Falls Sie diese Methode ausprobieren möchten, können Sie sie im Konstruktor der Komponente aufrufen (`this.createDemoFlight()`).

## Templates und die Datenbindung

Nachdem wir nun die Logik unserer Komponente in der Klasse `FlightSearchComponent` verstaut haben, können wir uns ihrem Template zuwenden. Es handelt sich dabei um die Datei `flight-search.component.html`.

Auf den ersten Blick handelt es sich hier um eine normale HTML-Datei. Neben HTML-Elementen kann sie jedoch auch sogenannte Datenbindungsausdrücke beinhalten. Damit gleicht Angular den Zustand der Komponente mit dem Zustand des Templates ab. Angular schreibt dazu beispielsweise Daten aus der Komponente in das Template oder übernimmt Eingaben in entsprechende Komponenteneigenschaften.

Eine erste Art von Datenbindungsausdruck haben Sie in Kapitel 1 im Rahmen der `AppComponent` bereits kennengelernt: Der Ausdruck

```
<h1>{{title}}</h1>
```

hat dort den Inhalt der Eigenschaft `title` ausgegeben.

Dieser Abschnitt geht auf die einzelnen von Angular unterstützten Datenbindungs- ausdrücke ein.

## Two-Way-Binding

Beim Einsatz von Formularen gilt es häufig, Eigenschaften aus der Komponente mit Eingabefeldern in der Anwendung abzugleichen: Die Werte der Eigenschaften sind also in Formularfelder zu übernehmen. Ändert der Anwender diese Felder, sind die neuen Werte in die jeweiligen Eigenschaften zurückzuschreiben. Diese Aufgabe übernimmt Angular mit sogenannten Two-Way-Bindings.

Wenn Sie mit einem Two-Way-Binding beispielsweise die Eigenschaft `from` aus unserer `FlightSearchComponent` an ein Eingabefeld binden wollen, müssen Sie in Angular folgende Schreibweise nutzen:

```
<input [(ngModel)]="from" name="from">
```



Kommt `input` innerhalb eines `form`-Elements zum Einsatz, muss es auch ein `name`-Attribut aufweisen. Details dazu finden sich in Kapitel 9.

Damit Sie auf den ersten Blick erkennen, dass es sich hier um ein Two-Way-Binding handelt, nutzt Angular eckige Klammern in Kombination mit runden. Die Community nennt diese Schreibkonvention auch *Banana-in-a-Box*. Zugegeben, dieser Einsatz von Sonderzeichen wirkt zunächst ein wenig seltsam. Allerdings hat sich das Angular-Team ganz bewusst für diese Schreibweise entschieden, um die Art der Datenbindung offensichtlich zu machen.

Bei `ngModel` handelt es sich um eine sogenannte *Direktive*. Direktiven sind von Angular bereitgestellte DOM-Erweiterungen, die Verhalten zur Seite hinzufügen. Im Fall von `ngModel` besteht dieses Verhalten im gewünschten Abgleich mit der angegebenen Eigenschaft. Gewissermaßen ist `ngModel` ein Experte für Eingabefelder: Es weiß, wie es die verschiedenen Eingabefelder – darunter Textfelder, Checkboxen, Radioboxen und Drop-down-Felder – mit den angegebenen Eigenschaften abgleichen kann.

Damit `ngModel` zur Verfügung steht, muss das `FormsModule` in unser `AppModule` importiert werden (siehe Beispiel 3-8).

*Beispiel 3-8: FormsModule bei AppModule registrieren*

```
// src/app/app.module.ts

[...]

// Diese Zeile einfügen:
import { FormsModule } from '@angular/forms';

@NgModule({
  imports: [
    // Diesen Eintrag hinzufügen:
    FormsModule,
    [...]
  ],
  declarations: [
    [...]
  ],
  providers: [],
  bootstrap: [
    AppComponent
  ]
})
export class AppModule { }
```



Two-Way-Data-Binding funktioniert nur mit ausgewählten Eigenschaften. Unter diesen ist `ngModel` die einzige, die Angular ab Werk zur Verfügung steht. Sie können jedoch eigene Eigenschaften, die Two-Way-Data-Binding unterstützen, entwickeln. Details dazu finden Sie in Kapitel 4.

## Property-Bindings

Ähnlich wie Two-Way-Bindings übernehmen Property-Bindings Eigenschaften aus der Komponente in das Markup. Auch nach dem Aktualisieren der Eigenschaften in der Komponente aktualisiert diese Binding-Art die Ausgabe. Allerdings schreibt sie Änderungen des Benutzers nicht mehr in die Komponente zurück. Deswegen könnte man hier auch von One-Way-Bindings sprechen.

Um solch ein Binding einzurichten, nutzen Sie eckige Klammern:

```
<button [disabled]="!from || !to">Search</button>
```

Das hier betrachtete Beispiel bindet den Ausdruck `!from || !to` an die DOM-Eigenschaft `disabled`. Der Ausdruck prüft, ob mindestens eine der beiden Eigenschaften leer ist. Das Beispiel deaktiviert somit die Schaltfläche, wenn keine Werte für diese Eigenschaften vorliegen.

Das Beispiel zeigt auch, dass Angular sich an standardmäßig vorherrschende DOM-Eigenschaften binden kann. Genau genommen, ist es aus Sicht von Angular egal, warum eine DOM-Eigenschaft existiert. Sowohl Standardeigenschaften als auch eigene Eigenschaften wie `ngModel` im letzten Abschnitt sowie DOM-Erweiterungen von anderen Bibliotheken lassen sich zusammen mit der Datenbindung nutzen.

Eine weitere Schreibweise für One-Way-Bindings sieht den bereits diskutierten Einsatz geschweifter Klammern vor:

```
<div>Es wurden {{ selectedFlight.length }} Flüge gefunden</div>
```

Damit platziert Angular eine Eigenschaft bzw. einen darauf basierenden Ausdruck mitten in der Seite.

## Direktiven

Wie bereits erwähnt, fügen Direktiven der Seite Verhalten hinzu. Dieses kann die Datenbindung unterstützen. Ein Beispiel dafür ist die Direktive `ngFor`, die eine Auflistung iteriert und pro Eintrag ein Stück HTML rendert (siehe Beispiel 3-9).

*Beispiel 3-9: Flight-Array mit ngFor iterieren*

```
<table class="table table-striped">
  <tr *ngFor="let flight of flights">
    <td>{{flight.id}}</td>
    <td>{{flight.from}}</td>
    <td>{{flight.to}}</td>
    <td>{{flight.date}}</td>
  </tr>
</table>
```

Im hier betrachteten Fall durchläuft `ngFor` sämtliche Flüge des Arrays `flights` aus der Komponente des vorherigen Abschnitts. Pro Flug rendert sie eine Tabellenzeile. Bitte beachten Sie, dass in Anlehnung an die `for-of`-Schleife in ECMAScript auch hier im Rahmen der Datenbindung das Schlüsselwort `of` zu verwenden ist.

Der vorangestellte Stern (\*ngFor) gibt darüber Auskunft, dass es sich beim Inhalt des aktuellen Elements um ein sogenanntes Template handelt. Damit sind hier HTML-Fragmente gemeint, die Angular zunächst gar nicht rendert und bei Bedarf einmal oder mehrere Male in die Seite einfügt.

Eine weitere Direktive, die sich hier anbietet, ist ngClass. Sie weist dem aktuellen Element zur bedingten Formatierung entsprechende Styles zu (siehe Beispiel 3-10).

*Beispiel 3-10: Bedingte Formatierung mit ngClass*

```
<table class="table table-striped">
  <tr *ngFor="let flight of flights"
      [ngClass]="{ 'active': flight === selectedFlight }">
    [...]
  </tr>
</table>
```

Somit erhält die Tabellenzeile mit dem gerade ausgewählten Flug die Klasse active. Dieser Style kann in der Datei *flight-search.component.scss* definiert werden:

```
.active {
  background-color:darkorange
}
```

In diesem Fall gilt der Style nur für die FlightSearchComponent. Um ihn global zur Verfügung zu stellen, ist er in die Datei *src/styles.scss* einzutragen.

Einen Überblick über weitere häufig verwendete Direktiven samt Alternativen dazu finden Sie in Tabelle 3-2.

Tabelle 3-2: Häufig verwendete Direktiven

Beispiel	Beschreibung
<tr *ngFor="let flight of flights"> <td>{{flight.id}}</td> </tr>	Iteriert über alle Flüge im Array flights und gibt pro Flug die ID aus.
<table *ngIf="flights.length > 0"> ... </table>	Blendet das Element ein, wenn der übergebene Ausdruck wahr (true bzw. truthy) ist.
<table *ngIf="flights.length > 0; else noFlights"> ... </table> <template #noFlights>Keine Flüge gefunden</template>	Das Schlüsselwort else verweist dazu auf den Namen eines Templates, das eingeblendet wird, wenn die Bedingung nicht erfüllt ist. Der Name des Templates wird mit einer Raute (#) als Präfix definiert. Diese Raute kommt jedoch nur bei der Deklaration des Namens und nicht bei dessen Verwendung zum Einsatz.
<tr [ngClass]="{ 'active': flight === selectedFlight }"> ... </tr>	Weist die Klasse active zu, wenn der Ausdruck flight === selectedFlight wahr (true bzw. truthy) ist.

Tabelle 3-2: Häufig verwendete Direktiven (Fortsetzung)

Beispiel	Beschreibung
<pre>&lt;tr [ngStyle]="{ 'background-color': bg }"&gt; ... &lt;/tr&gt;</pre>	Setzt die CSS-Eigenschaft background-color auf den Wert der Variablen bg.
<pre>&lt;tr [ngStyle]="{ 'background-color': (flight === selectedFlight) ? 'orange' : 'blue' }"&gt; ... &lt;/tr&gt;</pre>	Setzt die CSS-Eigenschaft background-color auf orange, wenn der Ausdruck flight === selectedFlight wahr (true) ist; ansonsten kommt der Wert blue zum Einsatz. Hierzu verwendet das Binding den aus JavaScript und anderen C-ähnlichen Sprachen bekannten ternären Operator.
<pre>&lt;tr [class.active]="flight === selectedFlight"&gt; ... &lt;/tr&gt;</pre>	Weist den Wert active zum Attribut class zu, wenn der übergebene Ausdruck wahr (true) ist. Bei dieser Kurzschreibweise werden der Attributname und der eventuell zuzuweisende Wert durch einen Punkt getrennt.
<pre>&lt;input [(ngModel)]="to" name="to"&gt;</pre>	Bindet die Eigenschaft to aus der Komponente mittels Two-Way-Binding an das Eingabefeld. Kommt das input-Element innerhalb eines form-Elements (<form>...</form>) zum Einsatz, erzwingt Angular das Vergeben eines Namens.

## Pipes

Ähnlich wie Direktiven unterstützen auch Pipes die Datenbindung. Sie sind in der Lage, Werte beim Binden zu verändern, und lassen sich somit unter anderem für das Formatieren von Werten nutzen. Zur Demonstration nutzt das folgende Beispiel die von Angular angebotene Pipe date zum Formatieren des Datums:

```
<td>{{flight.date | date:'dd.MM.yyyy HH:mm'}}</td>
```

Eine weitere standardmäßig vorhandene Pipe, die vor allem Entwicklerinnen und Entwicklern hilft, ist die Pipe json. Sie wandelt das gesamte Objekt in seine JSON-Repräsentation um. Somit können sie Objekte zum Testen ausgeben, ohne dafür eine Komponente oder Markup schreiben zu müssen:

```
<b>Basket</b>
<pre>{{ selectedFlight | json }}</pre>
```

Kapitel 6 geht näher auf Pipes ein und zeigt auch, wie sich eigene Pipes definieren lassen.

## Event-Bindings

Runde Klammern führen zu einer Bindung an Events. Dabei kann es sich sowohl um DOM-Events als auch um Erweiterungen von Frameworks wie Angular handeln. Das hier betrachtete Beispiel nutzt zwei Event-Bindings, um auf Mausklicks zu

reagieren. Das eine Event-Binding verknüpft die Schaltfläche *Search* mit der Komponentenmethode `search`:

```
<button (click)="search()" [disabled]="!from || !to">  
    Search  
</button>
```

Das andere Event-Binding ruft für einen der dargestellten Flüge die Methode `select` auf, um ihn als ausgewählten Flug vorzumerken:

```
<table class="table table-striped">  
    <tr *ngFor="let flight of flights"  
        [ngClass]="{ 'active': flight === selectedFlight }">  
  
        [...]  
        <td><a (click)="select(flight)">Select</a></td>  
    </tr>  
</table>
```



Verwenden Sie das folgende Styling in der Datei `src/styles.scss`, damit der Browser auch für Anchor-Tags ohne `href`-Attribut (z.B. bei `<a (click)="select(flight)">Select</a>`) den typischen Mauscursor für klickbare Links (Zeigefingersymbol) verwendet:

```
a {  
    cursor: pointer;  
}
```

3	Hamburg	Graz	26.12.2020 19:42	Select
4	Hamburg	Graz	26.12.2020 21:42	Select
5	Hamburg	Graz	27.12.2020 00:42	Select

## Das gesamte Template

Der Vollständigkeit halber zeigt Beispiel 3-11 das gesamte Template für die `FlightSearchComponent`, das wir in den vorangegangenen Abschnitten besprochen haben. Dabei fällt auf, dass die verwendeten Sonderzeichen, die bei ersten Schritten mit Angular durchaus gewöhnungsbedürftig sind, uns beim Erkennen der gewählten Datenbindungsart unterstützen und das Template somit nachvollziehbarer gestalten.

*Beispiel 3-11: Das gesamte Template für die Flugsuche*

```
<!-- src/app/flight-search/flight-search.component.html -->  
  
<h1>Flight Search</h1>  
  
<div class="form-group">  
    <label>From:</label>  
    <input [(ngModel)]="from" class="form-control">  
</div>
```

```

<div class="form-group">
  <label>To:</label>
  <input [(ngModel)]="to" class="form-control">
</div>

<div class="form-group">
  <button class="btn btn-default" (click)="search()" [disabled]="!from || !to">
    Search
  </button>
</div>

<table class="table table-striped">
  <tr *ngFor="let flight of flights"
      [ngClass]="{ 'active': flight === selectedFlight }">

    <td>{{flight.id}}</td>
    <td>{{flight.from}}</td>
    <td>{{flight.to}}</td>
    <td>{{flight.date | date:'dd.MM.yyyy HH:mm'}}</td>
    <td><a (click)="select(flight)">Select</a></td>
  </tr>
</table>

<b>Basket</b>
<pre>{{ selectedFlight | json }}</pre>

```

## Komponenten einbinden

Nachdem wir nun eine erste eigene Komponente geschaffen haben, müssen wir sie nur noch in unsere Anwendung einbinden. Damit die Angular-Anwendung unsere Komponente überhaupt berücksichtigen kann, muss sie in einem Angular-Modul deklariert werden. In unserem Fall handelt es sich dabei um das `AppModule`.

Diese Aufgabe sollte die CLI beim Generieren der Komponente schon übernommen haben. Aber zur Sicherheit lohnt es sich, das zu überprüfen. Öffnen Sie dazu die Datei `app.module.ts` und vergewissern Sie sich, dass die `FlightSearchComponent` unter `declarations` eingetragen ist (siehe Beispiel 3-12).

*Beispiel 3-12: Die `FlightSearchComponent` muss im `AppModule` deklariert werden.*

```

// src/app/app.module.ts

[...]
import { AppComponent } from './app.component';
[...]

@NgModule({
  imports: [
    FormsModule,
    HttpClientModule,
    BrowserModule
  ],
  declarations: [
    AppComponent,

```

```

SidebarComponent,
NavbarComponent,

// Unsere Komponente:
FlightSearchComponent
],
providers: [],
bootstrap: [
  AppComponent
]
})
export class AppModule { }

```

Danach können wir die Komponente im Template der AppComponent aufrufen (siehe Beispiel 3-13).

*Beispiel 3-13: Aufruf der FlightSearchComponent im Template der AppComponent*

```

<div class="wrapper">

  <div class="sidebar" data-color="white" data-active-color="danger">
    <app-sidebar-cmp></app-sidebar-cmp>
  </div>

  <div class="main-panel">
    <app-navbar-cmp></app-navbar-cmp>

    <div class="content">

      <!-- Alt: -->
      <!-- <h1>{{title}}</h1> -->

      <!-- Diese Zeile einfügen: -->
      <app-flight-search></app-flight-search>

    </div>
  </div>

</div>

```

## Anwendung ausführen und debuggen

Gratulation! Sie haben Ihre erste Angular-Anwendung geschrieben, und es ist nun an der Zeit, sie auszuführen.

### Anwendung starten

Zum Starten Ihrer Anwendung nutzen Sie die Angular CLI mit dem bereits diskutierten Befehl

```
ng serve -o
```

im Projekthauptverzeichnis. Nach dem Start des Entwicklungswebserver steht die Anwendung unter <http://localhost:4200> bereit (siehe Abbildung 3-3).

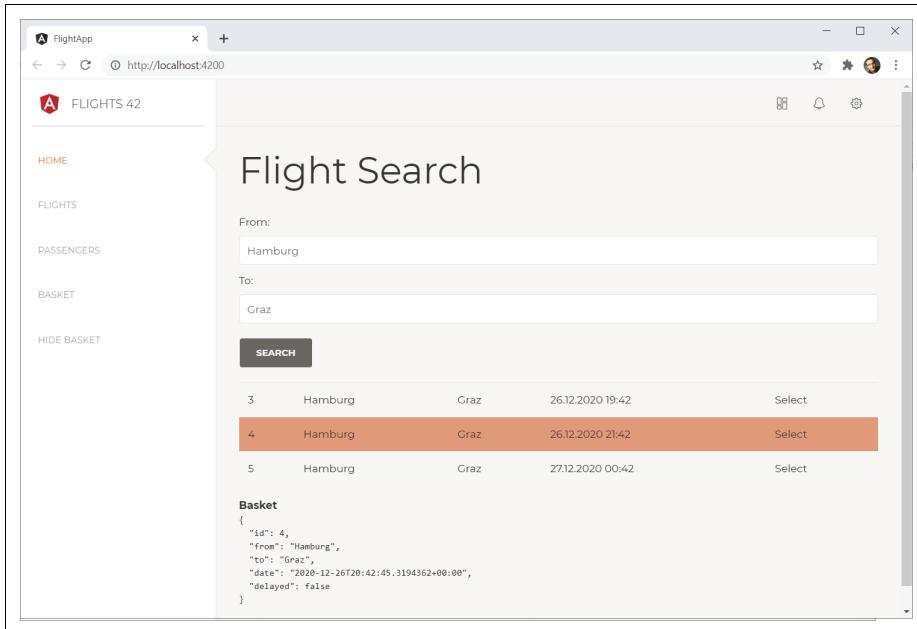


Abbildung 3-3: Ihre erste Komponente

## Fehler in der Entwicklerkonsole entdecken

Verhält sich die Anwendung nicht wie gewünscht, sollten Sie einen Blick auf die Konsole in den Entwicklerwerkzeugen (*F12* oder *Strg+Umschalt+I*) werfen. Hier finden Sie häufig Fehlermeldungen (siehe Abbildung 3-4).

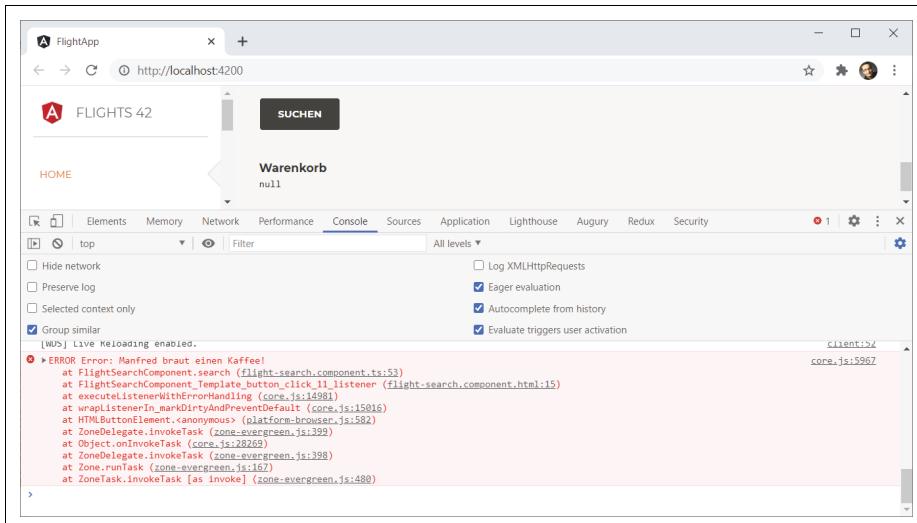


Abbildung 3-4: Fehler in der Entwicklerkonsole

Der Fehler in Abbildung 3-4 wurde zur Veranschaulichung mit der Anweisung

```
throw new Error('Manfred braut einen Kaffee!');
```

am Anfang der Methode search provoziert. In der Regel ist das jedoch nicht notwendig: Anwendungen weisen häufig auch ohne weiteres Zutun Bugs auf ;-).

Bitte beachten Sie die Hyperlinks, die Angular im Rahmen der Fehlermeldung ausgibt. Diese führen zu Zeilen in den betroffenen HTML- und TypeScript-Dateien, die beim Auftreten des Fehlers durchlaufen wurden.

## Die Anwendung im Browser debuggen

In Fällen, in denen Sie die Ursache des Fehlers nicht finden, können Sie auch den in den Browser integrierten JavaScript-Debugger einsetzen. Die Voraussetzung dafür ist, dass die CLI Metadaten für den Debugger – sogenannte *Source-Maps* – generiert hat. Beim Einsatz von ng serve ist das standardmäßig der Fall.

Bei Chrome finden Sie den Debugger in den Entwicklerwerkzeugen auf dem Registerblatt *Sources* (siehe Abbildung 3-5).

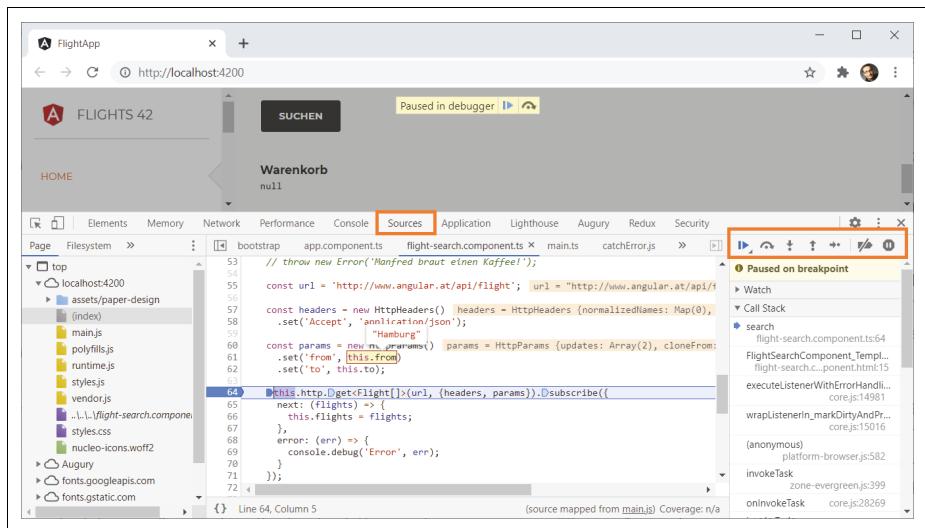


Abbildung 3-5: JavaScript-Debugger in Chrome

Hier können Sie Ihre Programmdateien öffnen und durch einen Klick auf eine Zeilennummer auf der linken Seite einen Break Point definieren. Zum Öffnen Ihrer Programmdateien empfiehlt sich die Tastenkombination *Strg+Umschalt+P*. Diese öffnet einen Dialog, mit dem Sie nach der gewünschten Datei suchen können. Geben Sie dazu einfach die ersten Buchstaben des Dateinamens ein.

Gelangt die Programmausführung zur Zeile mit dem Break Point, wird die Anwendung angehalten. Danach können Sie mit den Schaltflächen links oben die Ausfüh-

rung Schritt für Schritt fortsetzen und z.B. die aktuellen Werte Ihrer Variablen und Eigenschaften einsehen.

## Debuggen mit Visual Studio Code

Etwas komfortabler lässt sich der in Chrome integrierte Debugger über Visual Studio Code bedienen. Damit das möglich ist, müssen Sie das Visual-Studio-Code-Plug-in *Debugger for Chrome* installiert haben.

Zum Starten des Debuggers via Visual Studio Code benötigen Sie die Datei `.vscode/launch.json`. Falls sie noch nicht existiert, können Sie sie mit den folgenden Schritten einrichten:

1. Öffnen Sie eine beliebige `.ts`-Datei.
2. Wählen Sie in Visual Studio Code den Befehl *Run/Start Debugging* oder drücken Sie *F5*.
3. Falls Visual Studio Code Sie nach einer Umgebung (Environment) für das Debugging fragt, wählen Sie *Chrome* aus.
4. Visual Studio Code generiert nun eine Datei `launch.json` und zeigt diese an.
5. Korrigieren Sie in der Datei `launch.json` die angezeigte URL auf `http://localhost:4200`. Sie sollte in etwa wie die unter Beispiel 3-14 aussehen.

*Beispiel 3-14: launch.json zum Starten von Chrome via Visual Studio Code*

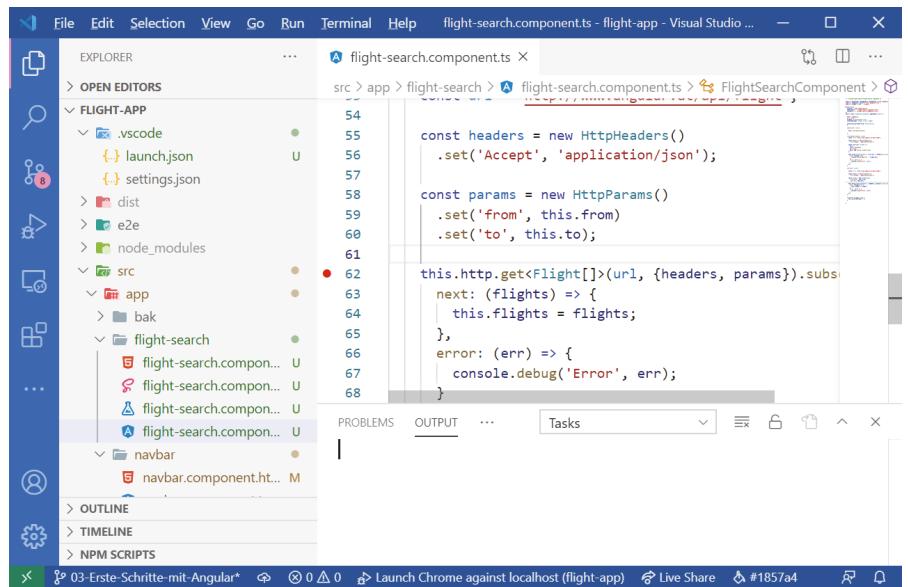
```
{  
  "version": "0.2.0",  
  "configurations": [  
    {  
      "type": "chrome",  
      "request": "launch",  
      "name": "Launch Chrome against localhost",  
      "url": "http://localhost:4200",  
      "webRoot": "${workspaceFolder}"  
    }  
  ]  
}
```

Wenn alle Stricke reißen, können Sie diese Datei auch manuell anlegen.

Um den Debugger nun via Visual Studio Code zu nutzen, sind die folgenden Schritte notwendig:

1. Starten Sie Ihre Anwendung wie gewohnt mit `ng serve`.
2. Erzeugen Sie direkt in Visual Studio Code durch einen Klick links neben einer Zeilennummer einen Break Point (siehe Abbildung 3-6).
3. Wählen Sie den Befehl *Run/Start Debugging* oder drücken Sie *F5*.
4. Nun öffnet sich Chrome.
5. Sobald der Programmfluss auf den Break Point stößt, hält der Debugger die Anwendung an.

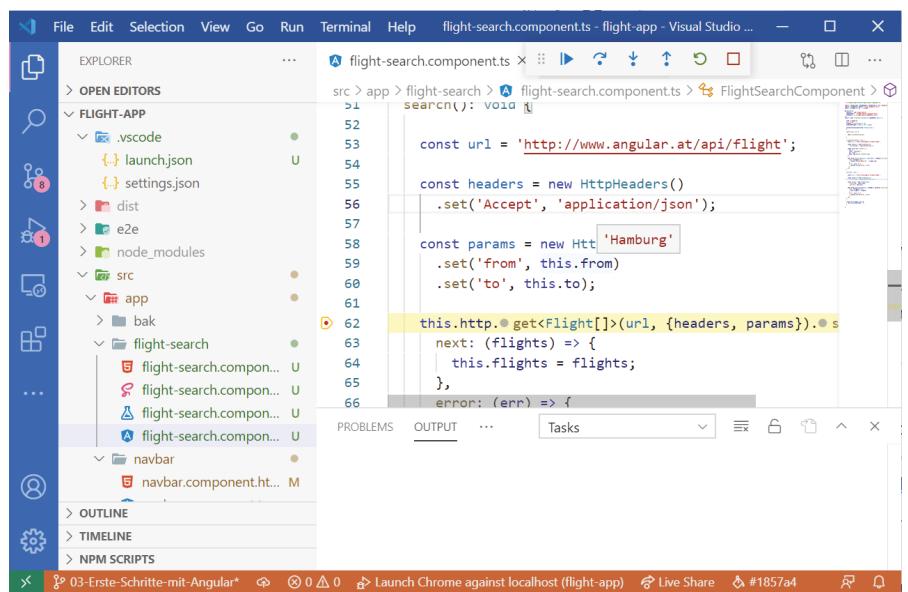
6. Sie können den Debugger jetzt direkt aus Visual Studio Code heraus steuern, die Ausführung Schritt für Schritt fortsetzen und die Werte von Variablen bzw. Eigenschaften einsehen (siehe Abbildung 3-7).



The screenshot shows the Visual Studio Code interface with the file `flight-search.component.ts` open in the editor. A red dot at the left margin of line 62 indicates a break point has been set. The code in line 62 is:

```
62 this.http.get<Flight[]>(url, {headers, params}).subscribe(flights => {
  this.flights = flights;
}, err => {
  console.debug('Error', err);
})
```

Abbildung 3-6: Break Point in Visual Studio Code (Zeile 62)



The screenshot shows the Visual Studio Code interface with the file `flight-search.component.ts` open in the editor. The code execution is paused at line 62, indicated by a yellow background and a yellow dot at the left margin. The code in line 62 is:

```
62 this.http.get<Flight[]>(url, {headers, params}).subscribe(flights => {
  this.flights = flights;
}, err => {
  console.debug('Error', err);
})
```

Abbildung 3-7: Debuggen mit Visual Studio Code

# Zusammenfassung

Angular-Anwendungen bestehen aus Komponenten. Hierbei handelt es sich um Klassen, die Informationen über Eigenschaften sowie das gewünschte Verhalten über Methoden anbieten. Dazugehörige Templates definieren, wie Angular die Komponenten darstellt. Mit Datenbindungsausdrücken stellen sie die Eigenschaften dar und verknüpfen Methoden mit UI-Ereignissen.

# Komponenten und Datenbindung

Eine Angular-Anwendung besteht aus Komponenten, die wiederum aus Komponenten bestehen. Dadurch ergibt sich ein Komponentenbaum. Somit kann eine komplexe Anwendung auf mehrere einfache, wiederverwendbare und testbare Teile heruntergebrochen werden.

Während wir in der Einführung zu Angular bereits eine erste Komponente beschrieben haben, zeigen wir Ihnen in diesem Kapitel, wie Sie eine solche Komponente in mehrere Komponenten zerlegen können, die über Datenbindung miteinander kommunizieren. Dabei handelt es sich auch um das Grundprinzip, aus dem sich der Komponentenbaum einer Anwendung ergibt.

## Datenbindung in Angular

Bevor wir Ihnen zeigen, wie Komponenten über Datenbindung kommunizieren, gehen wir in diesem Abschnitt darauf ein, wie Datenbindung in Angular überhaupt funktioniert.

### Rückblick auf AngularJS 1.x

Um die Architekturentscheidungen hinter der Datenbindung in Angular zu verstehen, lohnt sich ein Blick auf den Vorgänger, AngularJS 1.x. Hier konnte alles an alles gebunden werden. Um das zu veranschaulichen, zeigt Abbildung 4-1 zwei Datenmodelle sowie eine Direktive, die aneinanderggebunden wurden. Die Direktiven entsprechen in diesem Beispiel den heutigen Komponenten.

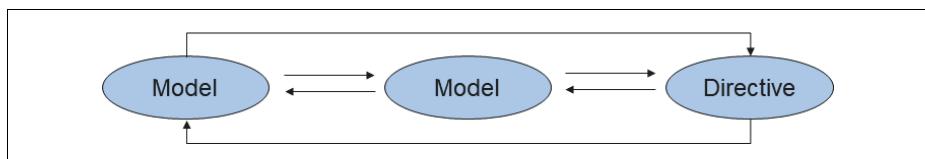


Abbildung 4-1: Zyklen in AngularJS 1.x

Durch die vielen wechselseitigen Abhängigkeiten konnte eine Änderung zu weiteren Änderungen führen, und diese konnten wiederum weitere Änderungen nach sich ziehen. Deswegen musste AngularJS 1.x immer im Kreis laufen. Dieses Im-Kreis-Laufend war auch als *Digest-Cycle* bekannt und war natürlich der Performance alles andere als zuträglich. Obwohl AngularJS 1.x in vielen Fällen schnell genug war, sind Programmiererinnen und Programmierer durch dieses Verhalten das eine oder andere Mal – abhängig von der Anwendungsarchitektur – in Performancefallen getappt, aus denen sie nur schwer wieder herausgekommen sind.

Bei Angular (ab Version 2) ist die Situation anders: Hier ist die Anwendung ein Komponentenbaum. Somit ergibt sich eine hierarchische Struktur, die das Einführen einiger einfacher Regeln ermöglicht (siehe Abbildung 4-2). Diese Regeln sind unter anderem der Schlüssel für die extrem gute Performance der Neuaufgabe. Die nächsten Abschnitte gehen darauf ein.

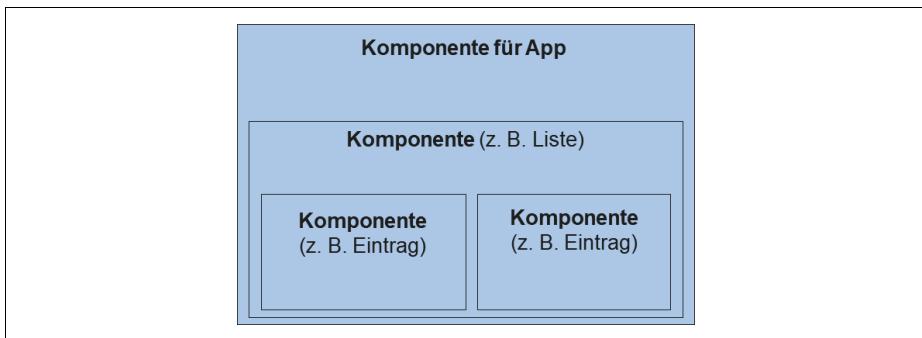


Abbildung 4-2: Hierarchische Struktur einer Angular-Anwendung

## Property-Binding

Für Property-Bindings gilt in Angular, dass sie immer nur Daten von einer Parent zu einer Child-Komponente weitergeben. Das bedeutet mit anderen Worten, dass Child-Komponenten im Rahmen der Datenbindung nicht ihren Parent verändern dürfen (siehe Abbildung 4-3). Die Daten fließen hier also von oben nach unten.

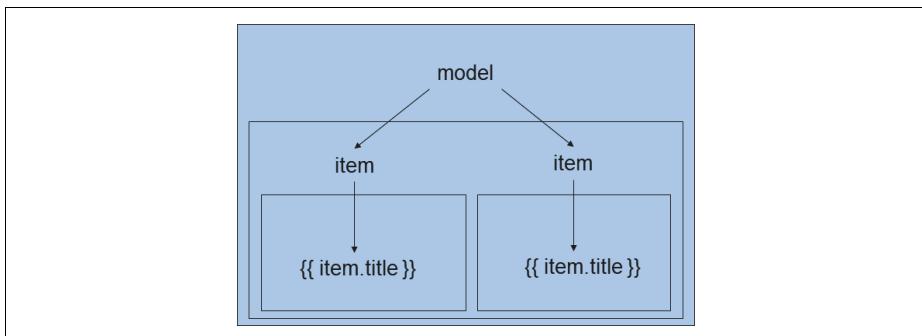


Abbildung 4-3: Die Daten fließen bei Property-Bindings von oben nach unten.

Der Abhängigkeitsgraph, der in AngularJS 1.x noch Zyklen enthalten konnte, ist nun ein Baum – also frei von Zyklen. Somit muss Angular auch nicht mehr im Kreis laufen. Vielmehr reicht es, den Baum ein einziges Mal zu traversieren, um ihn mit der UI abzulegen – oder anders ausgedrückt: Man benötigt lediglich einen einzigen Digest im Sinne von AngularJS 1.x. Da Angular für das Traversieren des Baums Code generiert, der sich gut von JavaScript-Engines optimieren lässt, ist dieser Vorgang äußerst schnell.



Um das Property-Binding zu optimieren, kann man das Framework wissen lassen, welcher Teil des Baums sich geändert hat. In diesem Fall beschränkt sich Angular auch nur auf den betroffenen Teilbaum. Da es sich hierbei um eine Optimierungstechnik handelt, die über die Grundlagen hinausgeht, finden Sie Informationen dazu erst in Kapitel 13.

## Event-Bindings

Event-Bindings definieren Handler für Ereignisse, die in Child-Komponenten auftreten. Ein Beispiel dafür ist das Click-Event aus der Einführung zu Angular. Die Schaltfläche löst es aus, behandelt wurde es jedoch im Handler der übergeordneten FlightSearchComponent.

Beim Auslösen eines Events kann die Child-Komponente einen Wert angeben, den der Handler im Parent erhält. Das dient der genaueren Beschreibung des aktuellen Ereignisses. Somit fließen bei Event-Bindings Informationen von unten nach oben (siehe Abbildung 4-4).

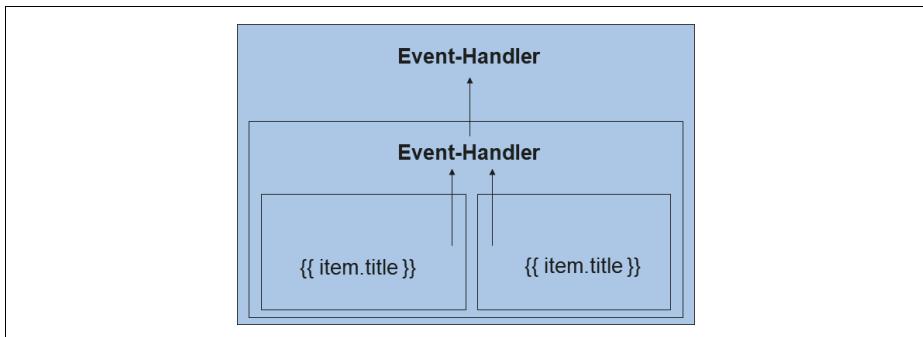


Abbildung 4-4: Die Daten fließen bei Event-Bindings von unten nach oben.

Das Schöne an Event-Bindings ist, dass sie sehr billig sind: Das Auslösen eines Event-Handlers besteht mehr oder weniger nur im Aufruf einer festgelegten Methode. Aus diesem Grund benötigt Angular dafür – um abermals die Nomenklatur aus AngularJS 1.x zu bemühen – keinen einzigen Digest.

Allerdings kann ein Event den Zustand der Anwendung verändern. Jemand klickt auf *Auswählen*, und plötzlich befindet sich ein neuer Flug im Warenkorb, oder je-

mand klickt auf *Suchen*, und ein paar Augenblicke später liegen neue Flüge vor, die es anzusehen gilt. Wie Angular damit umgeht, zeigt der nächste Abschnitt.

## Das Zusammenspiel von Property- und Event-Bindings

Damit die von Event-Handlern an den Komponenten durchgeführten Änderungen in der UI präsentiert werden, greifen bei Angular die beiden betrachteten Arten der Datenbindung ineinander. Das lässt sich durch einen Zustandsautomaten sehr gut veranschaulichen (siehe Abbildung 4-5).

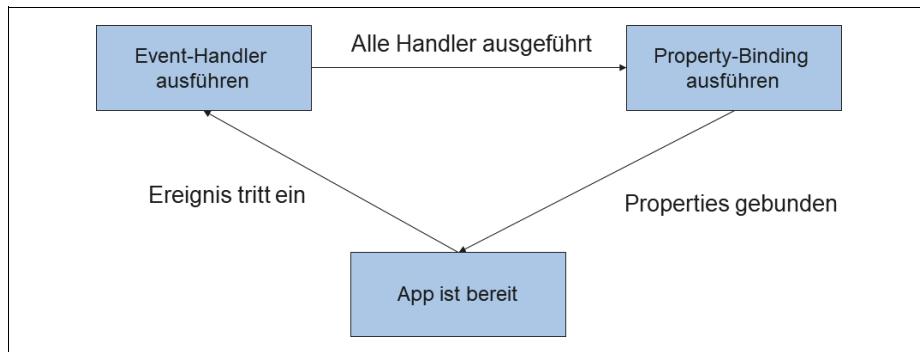


Abbildung 4-5: Datenbindung als Zustandsautomat

Nach dem Initialisieren der Anwendung ist sie bereit und wartet auf Ereignisse. Hierbei kann es sich um Benutzerereignisse (z.B. click oder keypress), Zeitereignisse (z.B. timeout) oder Datenereignisse (z.B. das Empfangen serverseitiger Daten) handeln. Um diese Ereignisse zu erkennen, klinkt sich Angular mit einem Mechanismus, der sich *zone.js* nennt, beim Start in sämtliche Event-Handler ein.

Tritt ein Ereignis auf, führt Angular die dafür registrierten Event-Handler aus. Ein Event-Handler kann weitere Events auslösen. Auf diese Weise kann eine Anwendung Informationen im Komponentenbaum nach oben transportieren.

Sind alle Event-Handler ausgeführt worden, führt Angular ein Property-Binding durch. Das ist notwendig, weil jedes Event gebundene Daten tendenziell verändern kann. Dazu traversiert es den gesamten Komponentenbaum ein einziges Mal. Wie schon erwähnt, ist dieser Vorgang bei Angular gut optimiert und somit in der Regel auch sehr schnell. Darüber hinaus existieren Optimierungsmöglichkeiten hierfür (siehe Kapitel 13).

Danach ist die Anwendung abermals bereit, und das Spiel beginnt von vorn. Es folgt also auf jede Eventphase ein Property-Binding. Somit findet genau ein Digest anstelle einer Vielzahl von Digests wie bei AngularJS 1.x statt. Wichtig ist hier auch, dass ein Property-Binding keine Events auslösen darf. Das würde ja einer Änderung des Parents gleichkommen, und so etwas ist – wie bereits gesagt – per Definitionem verboten. Somit verhindert Angular aufgrund seiner Architektur, dass mehr als ein Digest zu einem Zeitpunkt stattfindet.

## Bindings im Template

Wie wir schon in der Einführung zu Angular erwähnt haben, werden die beiden Arten von Bindings durch eine jeweils eigene Schreibweise im Template ausgedrückt. Während für Property-Bindings eckige Klammern zum Einsatz kommen, greift man bei Event-Bindings zu runden (siehe Beispiel 4-1). Damit ist auf den ersten Blick ersichtlich, welches Binding vorliegt.

*Beispiel 4-1: Template mit Bindings*

```
<button [disabled]="!from || !to" (click)="search()">  
    Search  
</button>  
  
<table>  
  
    <tr *ngFor="let flight of flights">  
        <td>{{flight.id}}</td>  
        <td>{{flight.date}}</td>  
        <td>{{flight.from}}</td>  
        <td>{{flight.to}}</td>  
        <td><a (click)="selectFlight(flight)">Select</a></td>  
    </tr>  
  
</table>
```

Sogar die Syntax mit den doppelt geschweiften Klammern ist genau genommen nichts anderes als eine Kurzschreibweise für ein Property-Binding. Im betrachteten Fall handelt es sich um ein Binding an die DOM-Property `text-content`, die den textuellen Inhalt eines Knotens widerspiegelt. Die Schreibweise

```
<td>{{ flight.id }}</td>
```

ist somit gleichbedeutend mit:

```
<td [text-content]="flight.id"></td>
```

Two-Way-Bindings wurden in diesem Beispiel ganz bewusst ausgespart. Warum, erklärt der nächste Abschnitt.

## Two-Way-Bindings

Wie wir schon erwähnt haben, kennt Angular Event-Bindings und Property-Bindings. Und beide Binding-Arten spielen zusammen. Da stellt sich nun die Frage, wo die Two-Way-Bindings geblieben sind. Gerade in formularbasierten Anwendungen muss man ja häufig den Formularzustand mit einem Objektmodell abgleichen.

Die Antwort hierauf lautet, dass ein Two-Way-Binding in Angular nichts anderes als eine Kombination aus einem Property-Binding und einem Event-Binding ist: Das Property-Binding transportiert den jeweiligen Wert von der Komponente ins

Eingabefeld, und bei einer Änderung durch den Benutzer transportiert das Event-Binding die Daten wieder zurück in die Komponente. In erster Näherung könnte also ein Two-Way-Binding wie folgt formuliert werden:

```
<input [ngModel]="from" (ngModelChange)="updateFrom($event)">
```

Dieses Beispiel nutzt `ngModel` als Property-Binding und das dazugehörige `ngModelChange` als Event-Binding. Letzteres wird bei jeder Änderung am Datenfeld aktiv und erhält über `$event` den geänderten Wert. Ändert der Benutzer beispielsweise *Hamburg* auf *Frankfurt* um, befindet sich in `$event` der neue Wert *Frankfurt*. Diesen Wert übergibt es an die Methode `updateFrom`, die sich in unserer Komponente befindet und die Ausgangsvariable `from` aktualisiert:

```
updateFrom(from: string): void {  
  this.from = from;  
}
```

Das Ganze lässt sich ein wenig abkürzen, indem der Code des Event-Handlers direkt im Event-Binding hinterlegt wird:

```
<input [ngModel]="from" (ngModelChange)="from = $event">
```

Noch kürzer wird es mit der bereits vorgestellten Banana-in-a-Box-Syntax:

```
<input [(ngModel)]="from">
```

Hierbei müssen Sie jedoch beachten, dass diese Grammatik nichts anderes ist als die Kurzschreibweise für die zuvor betrachtete explizite Form. Um daraus das Property-Binding `[ngModel]="from"` zu generieren, streift Angular einfach die runden Klammern ab.

Das Event-Binding wird hingegen durch Abstreifen der eckigen Klammern sowie durch Anhängen der Endung `Change` hergestellt. Aus `[(ngModel)]` wird somit `(ngModelChange)`. Daneben geht Angular davon aus, dass solche Events den geänderten Wert als `$event` erhalten. Diesen schreibt es in die Ausgangsvariable zurück.

Durch diese Architektur benötigt Angular selbst für das Aktualisieren von Two-Way-Bindings pro Änderung maximal einen Digest. Außerdem sollten Sie diese Konvention im Hinterkopf haben, wenn Sie eigene Komponenten schreiben. Sollten Sie ein Two-Way-Binding anbieten, müssen Sie sich genau an diese Konventionen halten. Der nächste Abschnitt geht darauf anhand eines Beispiels ein.

## Eigene Komponenten mit Datenbindung

Sie wissen jetzt, wie Datenbindung unter Angular funktioniert. In diesem Abschnitt zeigen wir, wie Sie eigene Komponenten schreiben, die über Bindings mit der Außenwelt kommunizieren.

## Überblick

Dazu wird eine Komponente geschaffen, die so einfach wie möglich, aber so komplex wie nötig ist, um die vorhin diskutierten Konzepte zu zeigen. Es handelt sich dabei um eine Komponente, die Flüge in Form von Karten präsentiert (siehe Abbildung 4-6).

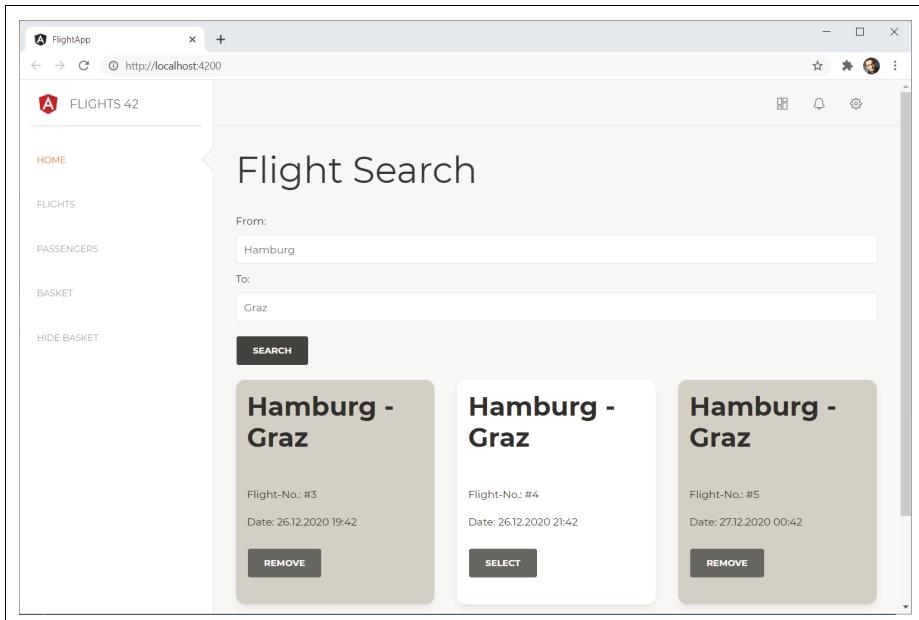


Abbildung 4-6: Die FlightCardComponent

Solche Karten sind derzeit sehr üblich, zumal sie ein flexibles (*responsive*) Design erlauben: Steht am Endgerät viel Platz zur Verfügung, kann eine Anwendung mehrere Karten nebeneinander anzeigen. Steht wenig Platz zur Verfügung, zeigt die Anwendung die Karten untereinander an.

## Vorbereitungen

Jede Karte kann ausgewählt werden. Wurde sie ausgewählt, erhält sie einen beigen Hintergrund, ansonsten einen weißen. Außerdem sollen alle ausgewählten Flüge im Warenkorb präsentiert werden. Dazu wird der Warenkorb auf ein Objekt abgeändert, das die IDs der Flüge auf einen boolean abbildet:

```
[...]  
export class FlightSearchComponent implements OnInit {  
  
  from = 'Hamburg';  
  to = 'Graz';  
  flights: Array<Flight> = [];  
  selectedFlight: Flight | null = null;
```

```

basket: { [key: number]: boolean } = {
  3: true,
  5: true
};

[...]

}

```

Im gezeigten Beispiel befinden sich von Anfang an die Flüge 3 und 5 im Warenkorb. Das soll das Ausprobieren unserer Anwendung ein wenig vereinfachen.

Der Datentyp von basket verdient unsere Aufmerksamkeit: { [key: number]: boolean } bedeutet, dass es sich hierbei um ein Objekt handelt, das Schlüssel vom Typ number auf Werte vom Typ boolean abbildet. Das Objekt wird also als Dictionary verwendet.

Genau genommen sind Objekte in JavaScript nichts anderes als Dictionaries. Normalerweise bilden sie die Namen von Eigenschaften auf deren Werte und die Namen von Methoden auf deren Implementierungen ab.

Eventuell fragen Sie sich, wie es möglich ist, dass die Schlüssel hier numbers sind, zumal in JavaScript die Schlüssel von Objekten ausschließlich Strings sind. Dazu müssen Sie sich vor Augen halten, dass wir es mit einem TypeScript-Compiler beim Kompilieren zu tun haben, und dieser erzwingt hier numbers. Zur Laufzeit haben wir es mit JavaScript zu tun. Dieses wandelt die numbers intern in Strings um.



Falls Ihnen die hier verwendete Schreibweise zu unübersichtlich ist, können Sie auch in einem vorgelagerten Schritt einen Typ für das Dictionary definieren und dann basket damit typisieren:

```

type NumberBooleanDict = { [key: number]: boolean };

[...]

export class FlightSearchComponent implements OnInit {
  [...]
  basket: NumberBooleanDict = {
    3: true,
    5: true
  }
  [...]
}

```

Um festzustellen, ob sich ein Flug im Warenkorb befindet, muss die Anwendung also nur prüfen, ob der Basket an der Stelle der *FlightId* truthy ist:

```
const inBasket = this.basket[7]; // 7 ist eine FlightId.
```

Zur Visualisierung des Warenkorbs kommt aus Gründen der Vereinfachung abermals die JSON-Pipe zum Einsatz:

```
{{ basket | json }}
```

Das Ganze gestaltet sich dann, wie in Abbildung 4-7 gezeigt.

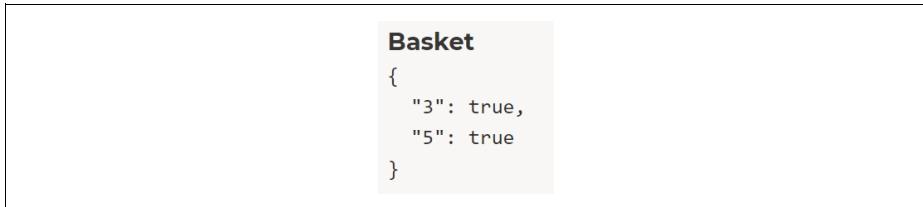


Abbildung 4-7: Ausgabe des Warenkorbs

## Eine Komponente mit Property-Bindings

Die hier besprochene Karte, deren Implementierung im nächsten Abschnitt folgt, soll über Property-Bindings zwei Informationen vom Parent übergeben bekommen: den anzuseigenden Flug und die Information, ob sie ausgewählt wurde. Für die erste Information weist die Komponente eine Eigenschaft `item` und für zweite Information eine Eigenschaft `selected` auf:

```
<div *ngFor="let f of flights">
  <app-flight-card [item]="f" [selected]="basket[f.id]">
    </app-flight-card>
</div>
```

Um alle gefundenen Flüge auszugeben, iteriert das betrachtete Beispiel über die Auflistung `flights` und gibt pro Eintrag eine Karte aus.

So können Sie sich das Einbinden einer Komponente wie den Aufruf einer Funktion vorstellen, die Parameter übergeben bekommt und ein Stück UI rendert. Eine andere Metapher für eine Komponente ist ein elektronisches Bauteil, z.B. ein Chip: Er ist über Eingänge mit der Außenwelt verdrahtet und bekommt auf diese Weise die nötigen Informationen (siehe Abbildung 4-8).

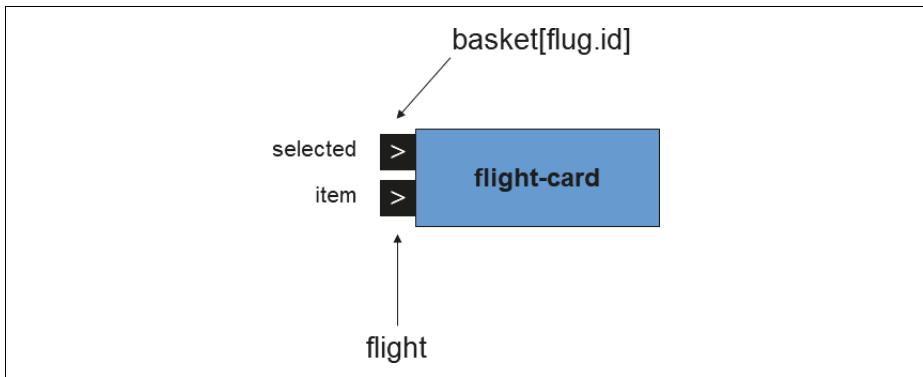


Abbildung 4-8: Die Komponente `flight-card` nimmt Informationen über Eigenschaften entgegen.

Im hier betrachteten Fall nimmt der Eingang `item` den jeweiligen Flug entgegen, und der Eingang `selected` bekommt den entsprechenden boolean aus dem Warenkorb.

Jetzt stellt sich natürlich die Frage, wie man mit Angular solche Eingänge darstellt. Der nächste Abschnitt geht darauf ein.

## Implementierung der Komponente mit Property-Bindings

Unsere Komponente wird wieder mit der Angular CLI generiert:

```
ng g c flight-card
```

Alternativ dazu lässt sich, wie in Kapitel 1 gezeigt, das Visual-Studio-Plug-in *Angular Schematics* dafür nutzen. Es richtet für diese Aufgabe im Kontextmenü der einzelnen Ordner einen Befehl *Angular: Generate a component* ein.

Die Implementierung unserer `flight-card` besteht zunächst mal aus einer Klasse mit einem Component-Dekorator (siehe Beispiel 4-2).

### Beispiel 4-2: Komponente mit Property-Bindings

```
// src/app/flight-card/flight-card.component.ts

import { Component, Input } from '@angular/core';
import { Flight } from '../flight';

@Component({
  selector: 'app-flight-card',
  templateUrl: './flight-card.component.html',
  styleUrls: ['./flight-card.component.scss']
})
export class FlightCardComponent {

  @Input() item: Flight | null = null;
  @Input() selected = false;

  select() {
    this.selected = true;
  }

  deselect() {
    this.selected = false;
  }
}
```

Der Dekorator erhält einen Selektor sowie einen Verweis auf ein Template. Den von der CLI generierten Konstruktor sowie die Implementierung von `OnInit` haben wir entfernt, da sie hier nicht benötigt werden.

Bis hierhin bietet diese Implementierung nichts Neues. Neu ist allerdings der `Input`-Dekorator. Er dekoriert sämtliche Eigenschaften, die die Komponente von ihrem Parent entgegennimmt.

Außerdem weist sie zwei Methoden auf, die ihr Template aufruft: select wählt die Karte aus, und deselect hebt diese Auswahl wieder auf.

Das Template dieser Komponente prüft zunächst, ob die Karte selektiert wurde. Ist dem so, erhält die Karte per ngClass eine entsprechende Formatierung (siehe Beispiel 4-3).

*Beispiel 4-3: Template der FlightCardComponent*

```
<!-- src/app/flight-card/flight-card.component.html -->

<div class="card" [ngClass]="{ 'active-card' : selected }">

  <div class="card-header">
    <h2 class="title">{{item?.from}} - {{item?.to}}</h2>
  </div>

  <div class="card-body">
    <p>Flight-No.: #{{item?.id}}</p>
    <p>Date: {{item?.date | date:'dd.MM.yyyy HH:mm'}}</p>
    <p>
      <button class="btn btn-default" *ngIf="!selected" (click)="select()">
        Select
      </button>

      <button class="btn btn-default" *ngIf="selected" (click)="deselect()">
        Remove
      </button>
    </p>
  </div>

</div>
```

Das Template gibt danach ein paar Daten des aktuellen Flugs aus. Bitte beachten Sie die Nutzung des Safe-Navigation-Operators (Fragezeichen): Statt item.id kommt hier zum Beispiel item?.id zum Einsatz. Das ist notwendig, weil die Eigenschaft item initial null ist und null.id im Strict Mode nicht erlaubt ist. Stattdessen veranlasst der Safe-Navigation-Operator Angular, die Navigation abzubrechen und null zurückzuliefern.

Das Styling für die Klasse active-card kann wieder lokal in die Datei *flight-card.component.scss* oder global in die Datei *styles.scss* eingetragen werden:

```
.active-card {
  background-color: rgb(204, 197, 185);
}
```

Diese Farbe wurde so gewählt, dass sie zum verwendeten Theming passt. Die anderen hier verwendeten Klassen werden von der eingebundenen Styling-Bibliothek Bootstrap definiert.

## Komponente registrieren und aufrufen

Auch diese Komponente muss bei einem Angular-Modul registriert werden. In unserem Fall handelt es sich um das AppModule. Der Aufruf von ng generate component oder die Nutzung des Plug-ins *Angular Schematics* in Visual Studio Code sollte diese Aufgabe automatisieren. Zur Sicherheit empfiehlt es sich jedoch, diesen Umstand zu prüfen (siehe Beispiel 4-4).

*Beispiel 4-4: Die FlightCardComponent wird im AppModule registriert.*

```
// src/app/app.module.ts

[...]
import { FlightCardComponent } from './flight-card/flight-card.component';

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

Danach erhält das gesamte Modul Zugriff auf die Komponente und lässt sich zur Präsentation gefundener Flüge in der FlightSearchComponent verwenden:

```
<div *ngFor="let f of flights">
  <app-flight-card [item]="f" [selected]="basket[f.id]">
  </app-flight-card>
</div>
```

Wie besprochen, erhält diese Komponente den aktuellen Flug und den Boolean aus dem Warenkorb. Die Anwendung sollte nun die gefundenen Flüge als Karten präsentieren.

Die Karten lassen sich auch über die präsentierten Schaltflächen aus- und abwählen. Ein kleines Problem fällt dabei allerdings auf: Angular aktualisiert die Eigenschaft basket und somit den präsentierten Warenkorb am Ende der Seite nicht. Hierzu müsste die FlightCardComponent ihren Parent, der den Warenkorb verwaltet, mit einem Ereignis benachrichtigen. Wie das geht, erläutert der nächste Abschnitt.



Falls Sie dieses Beispiel nachstellen, fällt Ihnen gegebenenfalls auf, dass die einzelnen Karten sehr viel Platz benötigen:

Um mehrere Karten nebeneinander zu präsentieren, kann man zum Spaltenlayout von Bootstrap greifen. Es ist für responsive Designs gedacht – also für Designs, die sich an unterschiedliche Auflösungen anpassen. Dazu unterteilt es eine Seite in zwölf gedachte Spalten, und die Anwendung weist jedem Element eine bestimmte Anzahl an Spalten zu. Dabei kann es zwischen sehr kleinen (*extra small*, `xs`), kleinen (`small`, `sm`), mittleren (`medium`, `md`), großen (`large`, `lg`) und sehr großen (*extra large*, `xl`) Bildschirmen unterscheiden. Beispiele für diese Größeneinheiten sind Handys (`xs`), Tablets (`sm` und `md`) sowie Laptops und Desktopgeräte (`lg` und `xl`). Hierbei handelt es sich jedoch nur um Näherungen, denn schlussendlich kommt es auf die zur Verfügung stehende Auflösung an.

Beispielsweise könnte man nun angeben, dass eine Karte bei sehr kleinen Geräten (`xs`) alle zwölf Spalten erhält, bei kleinen (`sm`) sechs, bei mittleren (`md`) sowie bei großen (`lg`) vier und bei sehr großen (`lg` und `xl`) drei der insgesamt zwölf Spalten. Somit werden je nach Auflösung eine bis vier Karten nebeneinander präsentiert. Hierzu sieht Bootstrap die nachfolgend verwendeten Klassen vor:

```
<div class="row">
  <div *ngFor="let f of flights" class="col-xs-12
  col-sm-6 col-md-4 col-lg-4 col-xl-3">
    <app-flight-card [item]="f"
      [selected]="basket[f.id]">
```

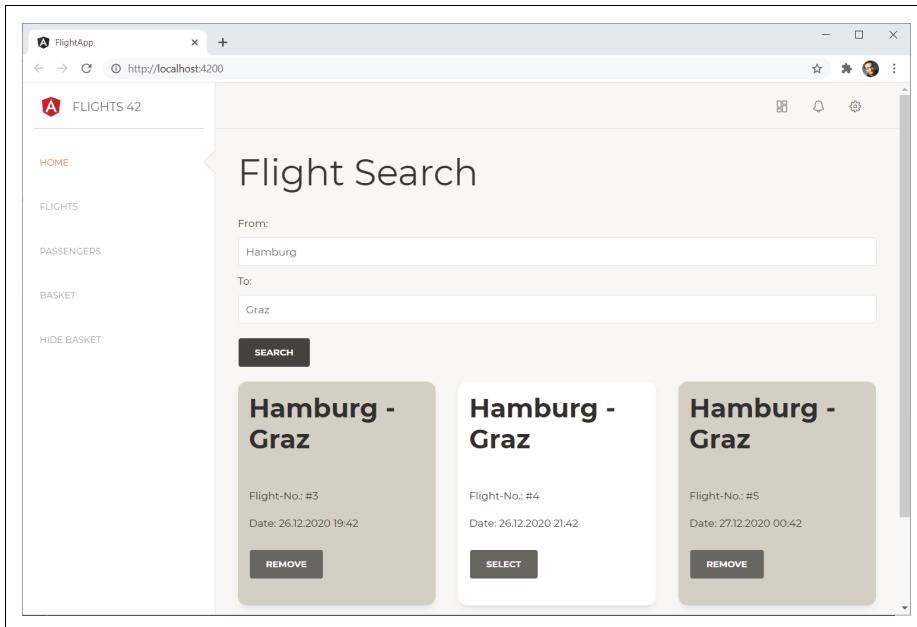
```

        </app-flight-card>
    </div>
</div>
```

Jede dieser Klassen, die mit dem Präfix col- eingeleitet werden, gibt für eine Auflösung die gewünschte Spaltenanzahl an. Beispielsweise bedeutet col-md-4, dass eine Karte bei einem mittleren Gerät vier der zwölf Spalten erhält.

Außerdem sind die einzelnen Spalten in einen Container, z.B. ein div, mit der Klasse row zu platzieren. Sie kümmert sich darum, dass bei Bedarf eine neue Zeile mit Flugkarten begonnen wird.

Das Ergebnis dieses Vorgehens sieht bei einem Bildschirm mit der Auflösung lg wie folgt aus:



## Komponenten mit Event-Bindings

Dieser Abschnitt erweitert die hier gezeigte FlightCardComponent um ein Ereignis selectedChange. Dieses Ereignis soll den Parent informieren, wenn die Karte aus- bzw. abgewählt wird. Beispiel 4-5 zeigt, wie sie verwendet werden soll:

*Beispiel 4-5: Nutzung einer Komponente mit Events*

```

<div *ngFor="let f of flights">
    <app-flight-card [item]="f"
        [selected]="'basket[f.id]'"
        (selectedChange)="basket[f.id] = $event">
    </app-flight-card>
</div>
```

Das Event `selectedChange` werden wir gleich einführen. Warten Sie bis dahin bitte mit dem hier gezeigten Aufruf, um Kompilierungsfehler zu vermeiden.

Man könnte sich diese eine Komponente als Funktion vorstellen, die einen Callback `selectedChange` übergeben bekommt. Immer wenn sie aus- bzw. abgewählt wird, ruft sie diesen Callback auf.

Die Metapher mit dem Chip passt hier noch besser: Ein Chip hat Ein- und Ausgänge, über die er mit seiner Umgebung verdrahtet wird. Die Ausgänge entsprechen den Events (siehe Abbildung 4-9). Im hier betrachteten Fall fließt der Wert `selected` über einen Ausgang zurück in den Warenkorb.

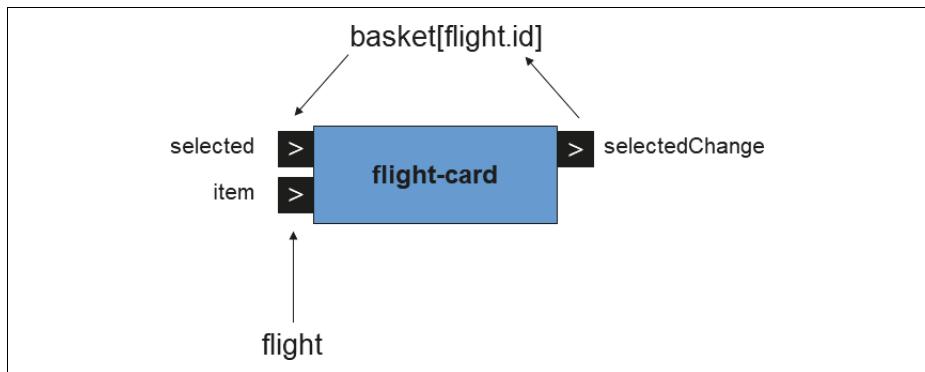


Abbildung 4-9: Komponente mit Eingängen (Properties) und einem Ausgang (Event)

### Implementierung der Komponente mit Event-Binding

Für das Event erhält die `FlightCardComponent` eine Eigenschaft `selectedChange`, die Sie mit `Output` dekorieren müssen (Beispiel 4-6).

#### Beispiel 4-6: Komponente mit Event

```
// src/app/flight-card/flight-card.component.ts

import { Component, Input, Output, EventEmitter } from '@angular/core';
import { Flight } from './flight';

@Component({
  selector: 'app-flight-card',
  templateUrl: './flight-card.component.html',
  styleUrls: ['./flight-card.component.scss']
})
export class FlightCardComponent {

  @Input() item: Flight | null = null;
  @Input() selected = false;
  @Output() selectedChange = new EventEmitter<boolean>();
```

```

    select() {
        this.selected = true;
        this.selectedChange.emit(true);
    }

    deselect() {
        this.selected = false;
        this.selectedChange.emit(false);
    }
}

```

Der Typ des Output ist per definitionem ein `EventEmitter`. Da es mehrere Typen mit diesem allgemeinen Namen gibt, sollten Sie sich vergewissern, dass Sie den Typ `EventEmitter` aus `@angular/core` importieren. Gerade beim Einsatz von Auto-Imports schlagen Entwicklungsumgebungen wie Visual Studio Code häufig den falschen Paketnamen vor.

Damit der `EventEmitter` den neuen Wert von `selected` veröffentlichen kann, wird er mit Boolean typisiert.

## Komponente aufrufen

Nach dieser Erweiterung können Sie mit dem Aufruf der `FlightCardComponent` einen Event-Handler für `selectedChange` festlegen. Beispiel 4-7 veranschaulicht die Nutzung der Komponente nach dem Einführen von `selectedChange`.

*Beispiel 4-7: Festlegen eines Event-Handlers für `selectedChange` beim Aufruf der `flight-card`*

```

<div class="row">
    <div
        *ngFor="let f of flights"
        class="col-xs-12 col-sm-6 col-md-4 col-lg-4 col-xl-3">

        <app-flight-card
            [item]="f"
            [selected]="basket[f.id]"
            (selectedChange)="basket[f.id] = $event">
        </app-flight-card>

    </div>
</div>

```

Die von Angular eingerichtete Variable `$event` beinhaltet den an `emit` übergebenen Wert, also `true` oder `false`. Die Anwendung sollte nun beim Aus- und Abwählen einer Karte den Warenkorb aktualisieren (siehe Abbildung 4-10).

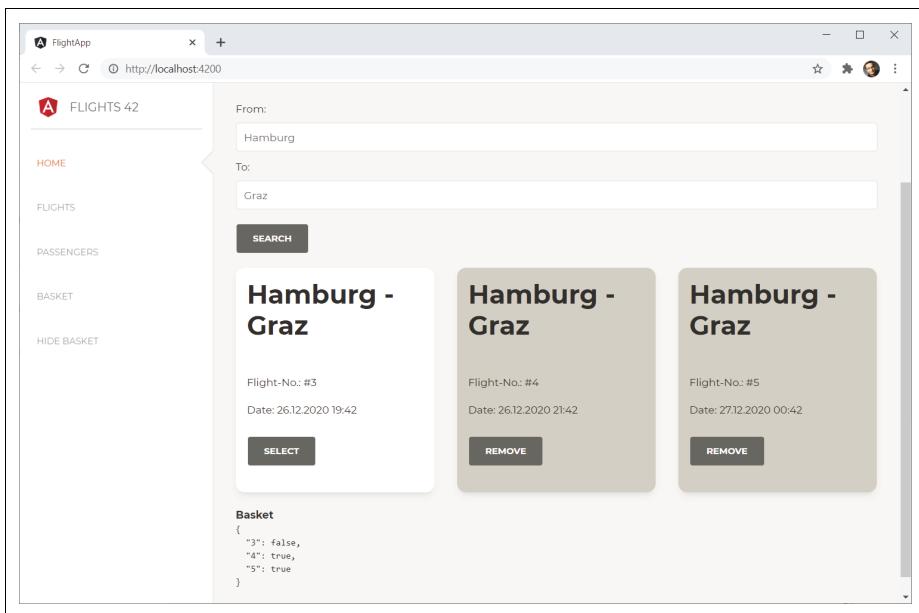


Abbildung 4-10: Der Warenkorb wird nun aktualisiert.

## Komponenten mit Two-Way-Bindings

Unsere Input/Output-Kombination für selected erfüllt sämtliche Konventionen für die verkürzte Banana-in-a-Box-Schreibweise, die wir im Abschnitt »Two-Way-Bindings« auf Seite 99 diskutiert haben: Das Event setzt sich aus dem Namen der Property sowie aus dem Suffix Change zusammen und veröffentlicht den geänderten Wert via \$event. Insofern spricht hier nichts gegen den Einsatz dieser komfortablen Grammatik (siehe Beispiel 4-8).

*Beispiel 4-8: Verkürzte Schreibweise für Two-Way-Data-Binding*

```
<div class="row">
  <div
    *ngFor="let f of flights"
    class="col-xs-12 col-sm-6 col-md-4 col-lg-4 col-xl-3">

    <app-flight-card
      [item]="f"
      [(selected)]="basket[f.id]">
    </app-flight-card>

  </div>
</div>
```

Hier zeigt sich auch der Nachteil dieser Abkürzung: Sie schreibt nach jeder Änderung den neuen Wert direkt in die Ausgangsvariable zurück. Wollte man hingegen zur Aktualisierung eine Methode anstoßen, müsste man das stattdessen explizit mit einem Event-Binding erledigen.

## Life-Cycle-Hooks

Eine Komponente unterliegt einem bestimmten Lebenszyklus: Sie wird irgendwann erzeugt, erhält Daten über Property-Bindings und wird irgendwann auch wieder zerstört. Letzteres ist z.B. der Fall, wenn die Bedingung eines umgebenden `ngIf` nicht mehr erfüllt ist.

Angular-Anwendungen können auf diese Stationen im Leben einer Komponente reagieren, indem sie Life-Cycle-Hooks implementieren.

## Ausgewählte Hooks

Angular bietet Hooks für verschiedene Zeitpunkte im Leben einer Komponente. In diesem Abschnitt stellen wir drei davon vor. In den folgenden Kapiteln führen wir anlassbezogen weitere ein.

Für jeden Life-Cycle-Hook definiert Angular ein Interface im Paket `@angular/core`, das eine Methode vorgibt (siehe Tabelle 4-1).

Tabelle 4-1: Ausgewählte Life-Cycle-Hooks

Interface	Methode	Beschreibung
<code>OnInit</code>	<code>ngOnInit</code>	Wird nach dem Initialisieren und somit nach dem ersten Ausführen der Property-Bindings aufgerufen.
<code>OnChanges</code>	<code>ngOnChanges</code>	Wird nach jedem Property-Binding aufgerufen. Der erste Aufruf erfolgt vor dem Aufruf von <code>OnInit</code> .
<code>OnDestroy</code>	<code>ngOnDestroy</code>	Wird aufgerufen, bevor Angular eine Komponente zerstört.

Um nun Hooks zu nutzen, implementiert die gewünschte Komponente die jeweiligen Interfaces und deren Methoden:

```
@Component({ [...] })
export class MyComponent implements OnChanges, OnInit {

    @Input() someData;

    ngOnInit() {
        [...]
    }

    ngOnChanges() {
        [...]
    }
}
```

## Experiment mit Life-Cycle-Hooks

Damit Sie sich mit Life-Cycle-Hooks vertraut machen können, beschreiben wir hier ein kleines Experiment. Es erweitert die weiter oben eingeführte FlightCard Component um die Hooks OnInit und OnChanges (siehe Beispiel 4-9):

*Beispiel 4-9: FlightCardComponent mit Life-Cycle-Hooks*

```
// src/app/flight-card/flight-card.component.ts

import { Component, Input, Output, EventEmitter, OnChanges, OnInit, SimpleChanges }
    from '@angular/core';

import { Flight } from '../flight';

@Component({
  selector: 'app-flight-card',
  templateUrl: './flight-card.component.html',
  styleUrls: ['./flight-card.component.scss']
})
export class FlightCardComponent implements OnInit, OnChanges {

  @Input() item: Flight | null = null;
  @Input() selected = false;
  @Output() selectedChange = new EventEmitter<boolean>();

  constructor() {
    console.debug('constructor', this.item);
  }

  ngOnInit() {
    console.debug('ngOnInit', this.item);
  }

  ngOnChanges(changes: SimpleChanges) {
    console.debug('ngOnChanges', this.item);

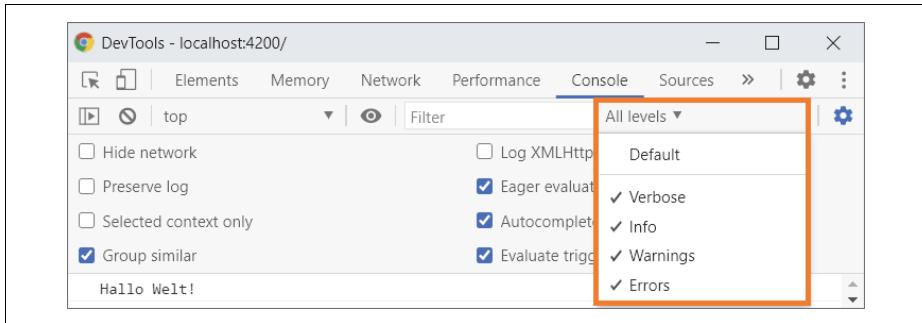
    if (changes.item) {
      console.debug('ngOnChanges: item');
    }
    if (changes.selected) {
      console.debug('ngOnChanges: selected');
    }
  }

  [...]
}

}
```



Bitte beachten Sie, dass Chrome standardmäßig Debug-Ausgaben auf der Konsole unterdrückt. Um sie einzublenden, müssen Sie das Level *Verbose* aktivieren:



Die beiden von den Hooks vorgegebenen Methoden geben ihren Namen sowie den aktuellen Flug in der Eigenschaft `item` auf der Konsole aus. Die Methode `ngOnChanges` prüft zusätzlich mit dem erhaltenen Parameter, welche Eigenschaften durch das letzte Property-Binding aktualisiert wurden, und notiert diese Erkenntnis auch auf der Konsole. Zum Vergleich gibt auch der Konstruktor den aktuellen Flug aus.

Lässt man nun diesen Code laufen, erhält man nach dem Suchen der Flüge die Ausgabe aus Abbildung 4-11.

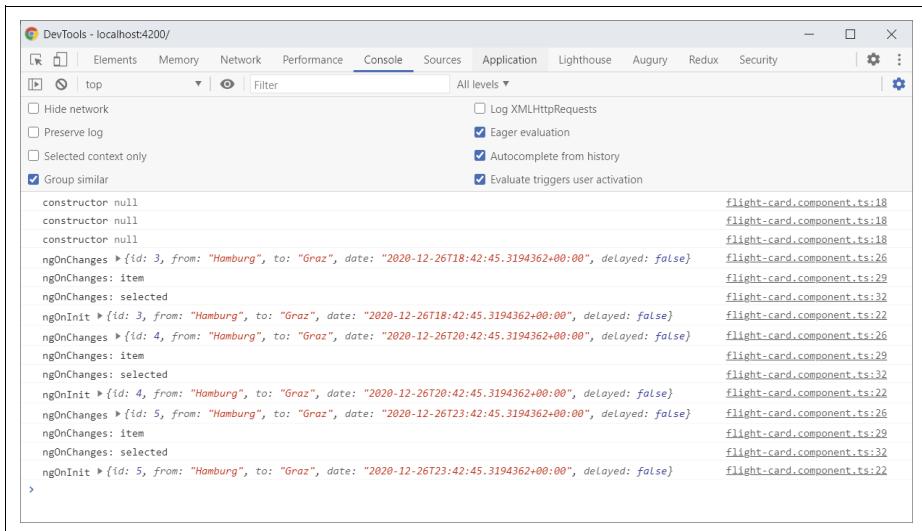


Abbildung 4-11: Debugging-Ausgaben bei der Initialisierung von drei `FlightCardComponent`-Instanzen

Diese Ausgabe zeigt, dass Angular bei der ersten Datenbindung pro Komponente zuerst `ngOnChanges` und dann erst `ngOnInit` aufruft. Das entspricht auch den Informationen aus Tabelle 4-1. Außerdem mag es auf den ersten Blick verwundern, dass im Konstruktor `item` noch null ist. Das liegt daran, dass der Konstruktor das

Erste ist, was JavaScript für eine Klasse ausführt. Zu diesem Zeitpunkt hat Angular noch gar keine Gelegenheit gehabt, ein Data-Binding auszuführen. Möchte eine Komponente also auf gebundene Daten zugreifen, muss sie dazu `ngOnInit` oder `ngOnChanges` nutzen.

Ändert man danach den Status einer Karte, erhält man nur den Aufruf von `ngOnChanges` (siehe Abbildung 4-12).

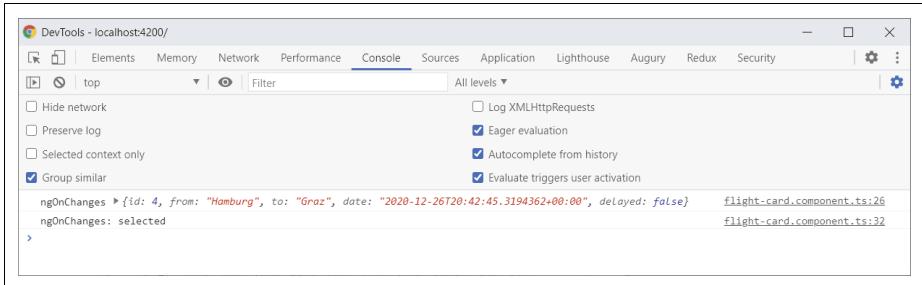


Abbildung 4-12: Debugging-Ausgabe nach Auswahl einer FlightCardComponent-Instanz

## Angular und Zyklen

Wie eingangs erwähnt, verbietet Angular Zyklen im Rahmen der Datenbindung: Event-Handler und Property-Bindings müssen streng sequenziell abgearbeitet werden. Diese Regel verhindert Performanceprobleme und verbessert die Nachvollziehbarkeit.

Life-Cycle-Hooks geben uns eine Möglichkeit, diese Regel – bewusst oder unbewusst – zu verletzen. Das Ergebnis ist ein Laufzeitfehler, der in der Praxis immer wieder zu Kopfzerbrechen führt. Deswegen möchten wir hier zeigen, wie Angular auf so einen Zyklus reagiert und was man dagegen machen kann.

Für dieses Experiment lösen wir im Life-Cycle-Hook `ngOnInit` das `selectedChange`-Event aus:

```
// src/app/flight-card/flight-card.component.ts
[...]
ngOnInit() {
  this.selectedChange.next(true);
  console.debug('ngOnInit', this.item);
}
[...]
```

Während Angular `ngOnInit` am Ende der Property-Bindings anstößt, führt das Auslösen eines Events zu einem erneuten Digest (siehe Abbildung 4-5). Somit herrscht hier ein Zyklus vor.

Glücklicherweise erkennt Angular diesen Zyklus und strafft uns mit einem `ExpressionHasBeenChangedAfterItHasBeenCheckedError` (siehe Abbildung 4-13).

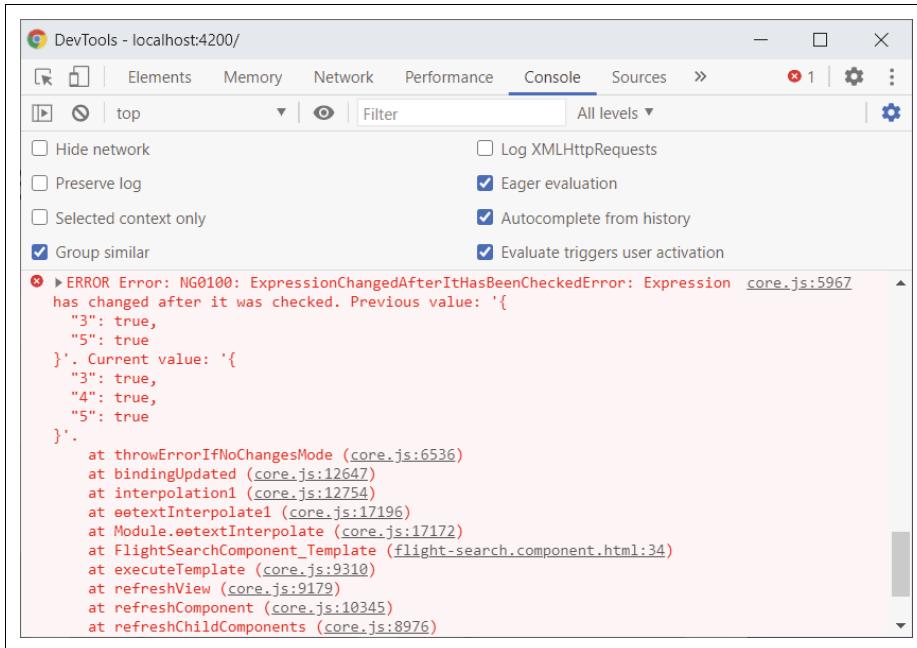


Abbildung 4-13: Laufzeitfehler als Folge eines Zyklus

Leider ist diese Fehlermeldung nicht unbedingt selbsterklärend. Angular beschwert sich damit darüber, dass sich unser basket-Objekt durch einen Zyklus geändert hat. Den Namen basket erwähnt Angular zwar nicht, aber die ausgegebene Objektstruktur lässt erkennen, dass es darum geht. Außerdem führt einer der gelisteten Hyperlinks in jene Zeile der `flight-search.component.html`, die die Eigenschaft basket per Datenbindung ausgibt. Die Fehlermeldung lässt übrigens auch erkennen, dass der Zyklus zum Hinzufügen des Eintrags mit dem Schlüssel 4 geführt hat.

Eventuell fragen Sie sich, wie uns Angular hier auf die Schliche gekommen ist. Die Antwort ist, dass im Debug-Modus Angular die Property-Bindings nicht nur einmal, sondern zweimal durchgeht. Der zweite Durchgang findet zur Kontrolle statt. Hier prüft Angular lediglich, ob sich die zuvor gebundenen Werte geändert haben. Falls ja, muss ein Zyklus vorliegen.

Kompilieren Sie Ihre Anwendung jedoch für die Produktion, z.B. mit `ng build`, deaktiviert Angular aus Performancegründen diese Prüfung. Das Angular-Team geht also davon aus, dass Sie solche Probleme bereits beim Debuggen und Testen entdecken und korrigieren.

Nun stellt sich die Frage, wie man dieses Problem loswird. Hierzu existiert eine Reihe einfacher Ansätze, die jedoch allesamt das eigentliche Problem nur kaschieren. Eine davon sieht vor, den Zyklus nach einem Time-out stattfinden zu lassen:

```
setTimeout(() => this.selectedChange.next(true), 0);
```

Auch wenn hier das zweite Argument ein Time-out von 0 Sekunden anfordert, reicht das, um die Ausführung von `this.selectedChange.next` nach hinten zu schieben. Angular führt also die diskutierte Prüfung *zuerst* aus, und erst *danach* wird der Zyklus angestoßen. Anders ausgedrückt: Das Time-out trickst Angular aus, sodass es den Zyklus nicht mehr erkennen kann.

Natürlich ergibt diese Vorgehensweise häufig wenig Sinn, zumal sie nichts am Zyklus ändert. Die einzige korrekte Lösung besteht darin, diese Änderung bereits weiter oben im Komponentenbaum stattfinden zu lassen. Reicht die Anwendung keine falschen Werte nach unten, müssen Child-Komponenten sie auch nicht korrigieren. In unserem Fall könnte z.B. schon die `FlightSearchComponent` die gewünschten Einträge im Basket hinterlegen.

Diese Strategie geht leider häufig mit einem umfangreichen Refactoring einher. Deswegen ist es sinnvoll, diese fundamentale und vom Angular-Team bewusst eingeführte Einschränkung von Anfang an im Hinterkopf zu haben.

## DateControl mit Life-Cycle-Hooks

Als Ergänzung zu dem Experiment aus dem letzten Abschnitt zeigen wir hier eine Komponente, die von einem Life-Cycle-Hook abhängig ist. Es handelt sich um eine einfache Datumskomponente, die auf den Namen `DateComponent` hört (siehe Abbildung 4-14).

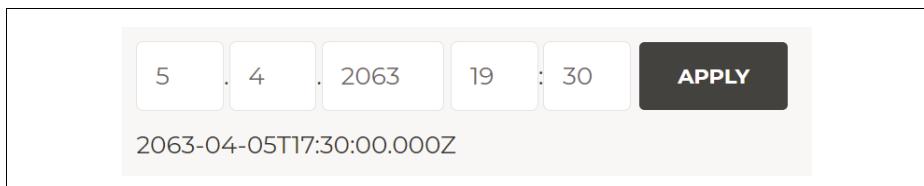


Abbildung 4-14: Einfache Datumskomponente

Um zu zeigen, dass das Two-Way-Binding auch die gebundene Eigenschaft in der Komponente aktualisiert, gibt dieses Beispiel sie darunter zusätzlich per Datenbindung aus. Der Unterschied um eine Stunde ergibt sich dadurch, dass auf dem verwendeten Rechner die mitteleuropäische Winterzeit eingestellt ist. Während die `DateComponent` diese Einstellung berücksichtigt, da sie das Date-Objekt von JavaScript nutzt, ist das bei der Ausgabe nicht der Fall. Diese verwendet, wie die Endung Z andeutet, die Universalzeit, die auch in Greenwich zum Einsatz kommt.

Die `DateComponent` wurde mit der Angular CLI generiert (`ng g c date`) und nimmt per Property-Binding ein Datum in Form eines ISO-Strings entgegen. Diesen zerlegt es in seine Einzelteile und bindet sie an die Eingabefelder. Da das Zerlegen des Datums immer dann zu erfolgen hat, wenn per Datenbindung ein neues Datum ankommt, kümmert sich der Life-Cycle-Hook `ngOnChanges` darum (siehe Beispiel 4-10):

*Beispiel 4-10: Die Datumskomponente zerlegt ein eingehendes Datum in ngOnChanges.*

```
import { Component, Input, OnInit, OnChanges, EventEmitter, Output, SimpleChanges }  
from '@angular/core';  
  
@Component({  
  selector: 'app-date',  
  templateUrl: './date.component.html',  
  styleUrls: ['./date.component.scss']  
})  
export class DateComponent implements OnInit, OnChanges {  
  
  @Input() date: string | null = null;  
  @Output() dateChange = new EventEmitter<string>();  
  
  day: number | null = null;  
  month: number | null = null;  
  year: number | null = null;  
  hour: number | null = null;  
  minute: number | null = null;  
  
  constructor() {  
    console.debug('date in constructor', this.date);  
  }  
  
  ngOnInit() {  
    console.debug('date in ngOnInit', this.date);  
  }  
  
  ngOnChanges(change: SimpleChanges) {  
    console.debug('date in ngOnChanges', this.date);  
  
    if (!this.date) {  
      return;  
    }  
  
    const date = new Date(this.date);  
    this.day = date.getDate();  
    this.month = date.getMonth() + 1;  
    this.year = date.getFullYear();  
    this.hour = date.getHours();  
    this.minute = date.getMinutes();  
  }  
  
  apply() {  
  
    if (!this.year || !this.month || !this.day || !this.hour || !this.minute) {  
      return;  
    }  
  
    const date = new Date(this.year, this.month - 1, this.day, this.hour,  
      this.minute);  
    this.dateChange.next(date.toISOString());  
  }  
}
```



Damit Ihre Komponente auf Änderungen an bestimmten Eigenschaften reagiert, können Sie als Alternative zu `ngOnChanges` auch mit TypeScript einen Setter einführen:

```
_date: string;
@Input() set date(value: string) {
  // Auf Wertänderung reagieren und Datum zerlegen.
  this._date = value;
}
```

Damit es hier keinen Namenskonflikt zwischen der Eigenschaft `date` und dem Setter `date` gibt, haben wir Erstere in `_date` umbenannt.

Die Methode `apply` kommt zum Einsatz, wenn der Benutzer seine Eingaben bestätigt. Sie erstellt ein neues Datum aus den Einzelteilen und stößt mit einem davon abgeleiteten ISO-String das Event `dateChange` an.

Das Template dieser Komponente besteht lediglich aus Textfeldern, die sich an die Teile des Datums binden. Außerdem weist es eine Schaltfläche auf, die die Methode `apply` aufruft (siehe Beispiel 4-11):

*Beispiel 4-11: Die Datumskomponente zerlegt ein eingehendes Datum mit `ngOnChanges`.*

```
<form class="form-inline">
  <input [(ngModel)]="day" name="day"
    maxlength="2" style="width:50px" class="form-control">

  <input [(ngModel)]="month" name="month"
    maxlength="2" style="width:50px" class="form-control">

  <input [(ngModel)]="year" name="year"
    maxlength="4" style="width:70px" class="form-control">
  &nbsp;
  <input [(ngModel)]="hour" name="hour"
    maxlength="2" style="width:50px" class="form-control">
  :
  <input [(ngModel)]="minute" name="minute"
    maxlength="2" style="width:50px" class="form-control">
  &nbsp;
  <input type="button" value="Apply" (click)="apply()" class="btn btn-default">
</form>
```

Damit Angular von der Existenz der Komponente erfährt, müssen Sie sich vergewissern, dass sie beim `AppModule` registriert wurde (siehe Beispiel 4-12).

*Beispiel 4-12: DateComponent registrieren*

```
// src/app/app.module.ts

[...]
import { DateComponent } from './date/date.component';

@NgModule({
  imports: [
```

```

        [...]
    ],
declarations: [
    [...]
    DateComponent
],
providers: [],
bootstrap: [
    AppComponent
]
})
export class AppModule { }

```

Um die Komponente zu testen, erhält die `FlightSearchComponent` ein weiteres Feld `date`:

```
date: string = (new Date()).toISOString();
```

Außerdem ruft ihr Template die `DateComponent` auf:

```
<div class="form-group">
    <label>Date:</label>
    <app-date [(date)]="date"></app-date>
    {{date}}
</div>
```

Das Ergebnis gestaltet sich wie das in Abbildung 4-15.

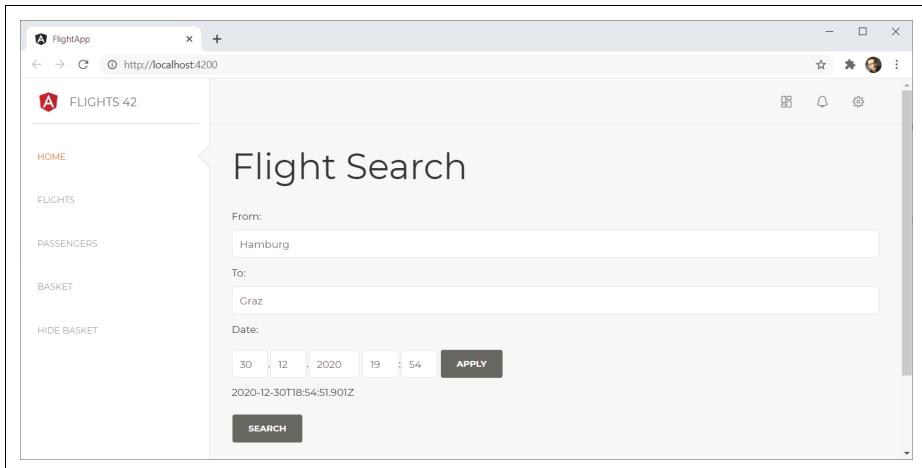


Abbildung 4-15: Die `DateComponent` in der `FlightSearchComponent`

# Zusammenfassung

Während bei Property-Bindings Daten im Komponentenbaum von oben nach unten fließen, ist es bei Event-Bindings genau andersherum: Hier fließen die Daten von unten nach oben. Um Zyklen zu vermeiden, führt Angular diese Bindings in zwei Phasen durch: Nach jedem Event kommen die Event-Bindings zur Ausführung, und danach kümmert Angular sich um die Property-Bindings, um die UI zu aktualisieren. Sogar Two-Way-Bindings sind streng genommen nur eine Kombination aus einem Property- und einem entgegengesetzten Event-Binding.

Eigene Komponenten können ebenfalls diese Bindings verwenden, um mit anderen Komponenten zu kommunizieren. Darüber hinaus benachrichtigt Angular jede Komponente mittels Life-Cycle-Hooks über bestimmte Ereignisse. Ein Beispiel dafür ist der Empfang initialer Daten oder neuer Daten über Property-Bindings.



# Services und Dependency Injection

Bis jetzt haben wir sämtliche Programmlogiken in Komponenten untergebracht. Möchte man jedoch dieselben Routinen in mehreren Komponenten nutzen, gilt es, sie an eine zentrale Stelle auszulagern. Hierfür bietet Angular das Konzept der *Services* an. Dabei handelt es sich häufig um wiederverwendbare Klassen.

Um verschiedene Konfigurationen zu unterstützen, lassen sich diese Services austauschen. Somit kann die Anwendung an die Geschäftsregeln verschiedener Kunden angepasst werden. Außerdem erhöht der Einsatz von Services die Testbarkeit. Beispielsweise könnte man für automatisierte Tests einen Service, der Daten via HTTP abruft, durch ein Gegenstück, das den HTTP-Zugriff nur simuliert, ersetzen.

Um diese Austauschbarkeit zu ermöglichen, fordern Komponenten die benötigten Services bei Angular an. Angular entscheidet aufgrund seiner Konfiguration, welche Ausprägung des Service geliefert wird. Hierbei ist auch von *Dependency Injection* die Rede.

Dieses Kapitel zeigt, wie Sie eigene Services schreiben und via Dependency Injection nutzen können.

## Ein erster Service

Unsere FlightSearchComponent kümmert sich derzeit direkt um das Abrufen von Flügen via HTTP. Allerdings ist es naheliegend, dass künftig auch weitere Komponenten die gleichen Serverzugriffe benötigen. Deswegen ist es üblich, solche Aufgaben in eigene Services auszulagern.

Genau das wird auch unsere erste Aufgabe in diesem Kapitel sein. Ähnlich wie Komponenten lassen sich Services mit der Angular CLI generieren. Führen Sie dazu den folgenden Befehl im Hauptverzeichnis Ihres Projekts aus:

```
ng generate service flight
```

Die Anweisung zum Generieren eines Service lässt sich auch abkürzen:

```
ng g s flight
```

Außerdem können Sie Services über das Kontextmenü eines Ordners in Visual Studio Code erzeugen, sofern Sie das Plug-in *Angular Schematics* installiert haben:

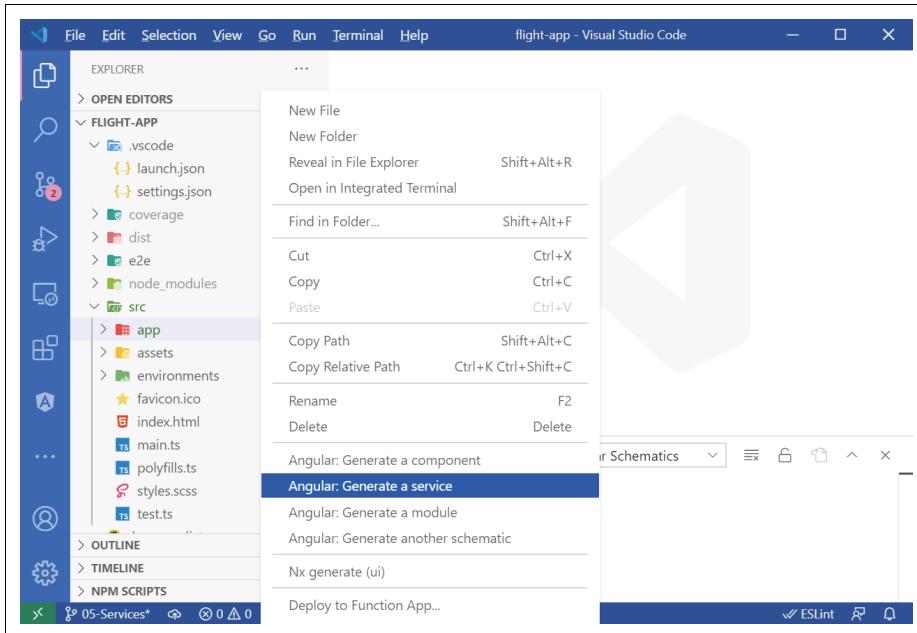


Abbildung 5-1: Service mit dem Plugin *Angular Schematics* erzeugen



In Kapitel 7 werden wir unsere Anwendung mit Angular-Modulen besser strukturieren und dabei auch unsere Services und Komponenten in Unterordner verschieben.

Dieser Befehl veranlasst die CLI, zwei Dateien zu generieren (siehe Abbildung 5-2).

```
ng generate service flight
CREATE src/app/flight.service.spec.ts (357 bytes)
CREATE src/app/flight.service.ts (135 bytes)
```

Abbildung 5-2: Service mit der CLI generieren

Während die Datei *flight.service.ts* das Grundgerüst unseres neuen FlightService beinhaltet, findet sich in der Datei *flight.service.spec.ts* das Grundgerüst für einen dazugehörigen Unit-Test. Letzteren wollen wir vorerst jedoch nicht genauer betrachten. Wir kommen in Kapitel 12 darauf zurück.

Beispiel 5-1 zeigt das generierte Grundgerüst für unseren FlightService.

### Beispiel 5-1: Grundgerüst des generierten FlightService

```
// src/app/flight.service.ts

import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class FlightService {

  constructor() { }
}
```

Die Konfiguration von Services nennt man auch *Provider* oder *Serviceprovider*. Wird der Service wie hier über Eigenschaften von `Injectable` konfiguriert, ist auch von *Tree-Shakable Provider* die Rede. Der Name röhrt daher, dass solche Provider gut mit einer Optimierungstechnik namens Tree-Shaking zusammenspielen. Diese Technik entfernt beim Kompilieren alle nicht benötigten Framework-Bestandteile und trägt somit zu einer Minimierung der Bundles bei. Die Angular CLI kümmert sich übrigens automatisch um diese Aufgabe, wenn Sie Ihre Bundles mit `ng build` bauen lassen.

Beim gezeigten Beispiel handelt es sich lediglich um eine Klasse mit einem `Injectable`-Dekorator. Aufgrund dieses Dekorators weiß Angular, dass wir diese Klasse als Service nutzen wollen.

Die Eigenschaft `providedIn` gibt den Scope des Service an. Anders ausgedrückt: `providedIn` sagt uns, wo in der Anwendung der Service zur Verfügung steht. In der Regel werden Sie auf die folgenden beiden Optionen stoßen:

*root (String)*

Der String `root` gibt an, dass der `FlightService` in der gesamten Anwendung zur Verfügung steht. Man spricht hierbei auch vom *Root-Scope*. Sie werden diese Option in den meisten Fällen wählen.

*Verweis auf ein lazy Angular-Modul*

Eine Anwendung kann angewiesen werden, ein Angular-Modul erst bei Bedarf in den Browser zu laden. Hierbei ist von *lazy loading* die Rede. Verweist `providedIn` auf so ein Modul, wird der Service gemeinsam mit diesem Modul geladen und kann deswegen auch nur innerhalb dieses Moduls genutzt werden. Weitere Informationen hierzu finden Sie in Kapitel 13.



Es ergibt übrigens keinen Sinn, `providedIn` auf ein Modul, das nicht per Lazy Loading bezogen wird, verweisen zu lassen. Diese Module, die von Anfang an zur Verfügung stehen, teilen sich nämlich den Root-Scope. Insofern hätte dieses Vorgehen denselben Effekt wie `providedIn: root`.

Lassen Sie uns nun dem `FlightService` eine Methode `find` zum Suchen nach Flügen spendieren (siehe Beispiel 5-2).

*Beispiel 5-2: FlightService mit der Methode find*

```
// src/app/flight.service.ts

import { HttpClient, HttpHeaders, HttpParams } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import { Flight } from './flight';

@Injectable({
  providedIn: 'root'
})
export class FlightService {

  constructor(private http: HttpClient) { }

  find(from: string, to: string): Observable<Flight[]> {
    const url = 'http://demo.ANGULARArchitects.io/api/flight';

    const headers = new HttpHeaders()
      .set('Accept', 'application/json');

    const params = new HttpParams()
      .set('from', from)
      .set('to', to);

    return this.http.get<Flight[]>(url, {headers, params});
  }
}
```

Im Wesentlichen entspricht diese neue Methode dem Aufbau der Methode search, die wir in Kapitel 3 direkt in der FlightSearchComponent platziert haben. Beachten Sie bitte die folgenden Punkte:

- Der FlightService lässt sich den HttpClient injizieren. Services können demnach auch weitere Services via Dependency Injection anfordern.
- Die Methode find liefert das Ergebnis von this.http.get als Observable<Flight> zurück. Das bedeutet, dass der Aufrufer von find bei diesem Observable die Methode subscribe aufrufen muss, um die abgerufenen Flüge in Empfang zu nehmen.

Nun können wir unseren FlightService in der FlightSearchComponent nutzen (siehe Beispiel 5-3).

*Beispiel 5-3: FlightService in FlightSearchComponent verwenden*

```
// src/app/flight-search/flight-search.component.ts

import { Component, OnInit } from '@angular/core';
import { Flight } from '../flight';
import { FlightService } from '../flight.service';

@Component({
  selector: 'app-flight-search',
  templateUrl: './flight-search.component.html',
  styleUrls: ['./flight-search.component.css']
})
```

```

        styleUrls: ['./flight-search.component.scss']
    })
export class FlightSearchComponent implements OnInit {

    from = 'Hamburg';
    to = 'Graz';
    flights: Array<Flight> = [];
    selectedFlight: Flight | null = null;

    basket: { [key: number]: boolean } = {
        3: true,
        5: true
    };

    constructor(private flightService: FlightService) {}

    ngOnInit(): void {}

    search(): void {

        this.flightService.find(this.from, this.to).subscribe({
            next: (flights) => {
                this.flights = flights;
            },
            error: (err) => {
                console.debug('Error', err);
            }
        });
    }

    select(f: Flight): void {
        this.selectedFlight = f;
    }
}

```

Die aktualisierte `FlightSearchComponent` lässt sich den `FlightService` in den Konstruktor injizieren. Die Methode `search` verwendet diesen `FlightService` zum Abrufen von Flügen.



Services sind immer Singletons. Das heißt, Angular erzeugt davon nur eine Instanz pro Scope. Da wir bis jetzt nur den Root-Scope verwendet haben, existiert in der gesamten Anwendung nur eine einzige Instanz unseres Service.

Der Abschnitt »Einen Service lokal registrieren« auf Seite 135 sowie das Kapitel 13 zeigen, unter welchen Umständen sich weitere Scopes ergeben sowie welche Auswirkungen das auf Services hat.

Der zuvor injizierte `HttpClient` wird nicht mehr benötigt. Deswegen wurde seine Verwendung aus der `FlightSearchComponent` ersatzlos entfernt. Das betrifft auch die

in Kapitel 3 gezeigte Demomethode `createDemoFlight`, die zur Veranschaulichung einen neuen Flug erzeugt.

Gratulation! Sie haben Ihren ersten Service mit wiederverwendbarer Logik geschrieben und in einer Komponente verwendet. Die nächsten Abschnitte gehen auf weitere Details zu diesem Thema ein.

## Services austauschen

Ein wichtiges Merkmal von Services ist, dass sie sich gegen andere Services austauschen lassen. Beispielsweise lässt sich Angular per Konfiguration anweisen, allen Komponenten, die per Dependency Injection einen `FlightService` anfordern, einen `DummyFlightService` zu spendieren:

```
@Component({ [...] })
export class FlightSearchComponent implements OnInit {
    [...]
    constructor(private flightService: FlightService) {
        if (flightService instanceof DummyFlightService) {
            console.debug('Eigentlich bin ich ein DummyFlightService');
        }
    }
    [...]
```

In diesem Fall tritt der `FlightService` als sogenanntes Token auf. Das Token ist der Typ, den der Konsument anfordert. Der Service ist hingegen die Implementierung, die Angular tatsächlich liefert.

Damit das Austauschen von Services nicht zu Laufzeitfehlern führt, muss der Ersatzservice die gleichen Methoden und Eigenschaften aufweisen. Diese Kompatibilität lässt sich durch den Einsatz abstrakter Basisklassen als Token erzwingen.

Um das zu demonstrieren, erweitern wir hier den Aufbau unseres Beispiels, wie in Abbildung 5-3 gezeigt.

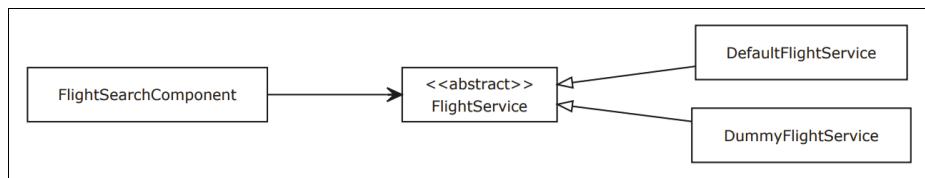


Abbildung 5-3: Abstrakte Basisklasse mit zwei Serviceausprägungen

Diese Änderung hat die folgenden Charakteristiken:

- Die `FlightSearchComponent` fordert nach wie vor einen `FlightService` via Dependency Injection an.
- Der Typ `FlightService` wird hier als Token verwendet. Es handelt sich dabei nun um eine abstrakte Basisklasse, die die abstrakte Methode `find` vorgibt.

- Die beiden konkreten Ausprägungen `DefaultFlightService` und `DummyFlightService` implementieren die abstrakte Basisklasse `FlightService` und somit auch `find`.

Hierdurch ist sichergestellt, dass die beiden konkreten Ausprägungen jene Struktur aufweisen, die sich der `FlightService` erwartet.



Leider lassen sich Interfaces nicht als Tokens verwenden. Der Grund dafür liegt im Verhalten des TypeScript-Compilers. Dieser nutzt Interfaces lediglich beim Kompilieren, ohne im erzeugten JavaScript einen Typ dafür zu erzeugen. Deswegen existieren Interfaces zur Laufzeit nicht mehr. Tokens werden jedoch zur Laufzeit benötigt, damit die Anwendung den konfigurierten Service anfordern kann.

Der `DefaultFlightService` und der `DummyFlightService` lassen sich wie gewohnt mit der Angular CLI erzeugen:

```
ng generate service default-flight
ng generate service dummy-flight
```

Als Alternative zur direkten Verwendung der CLI können Sie natürlich auch hier wieder das Plug-in *Angular Schematics* in Visual Studio Code verwenden.

Wie besprochen, wird der `FlightService` in eine abstrakte Basisklasse abgeändert (siehe Beispiel 5-4).

#### *Beispiel 5-4: Abstrakter FlightService als Basisklasse*

```
// src/app/flight.service.ts

import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import { DefaultFlightService } from './default-flight.service';
import { Flight } from './flight';

@Injectable({
  providedIn: 'root',
  // Diese Umleitung hinzufügen:
  useClass: DefaultFlightService
})
// Klasse als abstrakt kennzeichnen:
export abstract class FlightService {

  // Die Methode find ist nun abstrakt:
  abstract find(from: string, to: string): Observable<Flight[]>;
}
```

Die gesamte Klasse ist nun als abstrakt gekennzeichnet (`export abstract class FlightService`). Die Methode `find` bekommt ebenfalls die Markierung `abstract`, und ihre Implementierung wird entfernt.

Angular muss nun jedoch wissen, welche konkrete Implementierung heranzuziehen ist, wenn sich eine Komponente wie die `FlightSearchComponent` einen `Flight`

Service injizieren lassen möchte. Diese Information wurde dem Dekorator `Injectable` über die Eigenschaft `useClass` hinzugefügt. Sie verweist nun auf den `DefaultFlightService`.

Als Standardimplementierung gestaltet sich der `DefaultFlightService` wie die ursprüngliche Variante des `FlightService` (siehe Beispiel 5-5).

*Beispiel 5-5: Standardimplementierung für den FlightService*

```
// src/app/default-flight.service.ts
```

```
import { HttpClient, HttpHeaders, HttpParams } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import { Flight } from './flight';
import { FlightService } from './flight.service';

@Injectable({
  providedIn: 'root'
})
export class DefaultFlightService implements FlightService {

  constructor(private http: HttpClient) { }

  find(from: string, to: string): Observable<Flight[]> {
    const url = 'http://demo.ANGULARArchitects.io/api/flight';

    const headers = new HttpHeaders()
      .set('Accept', 'application/json');

    const params = new HttpParams()
      .set('from', from)
      .set('to', to);

    return this.http.get<Flight[]>(url, {headers, params});
  }
}
```

Beachten Sie bitte, dass der `DefaultFlightService` den abstrakten `FlightService` implementiert. Somit erzwingt TypeScript das Vorhandensein der vorgegebenen Methode `find`.



Falls Sie Erfahrung mit anderen Sprachen wie Java oder C# haben, werden Sie sich nun eventuell fragen, warum die abstrakte Basisklasse implementiert (`implements`) und nicht vererbt (`extends`) wird. TypeScript unterstützt tatsächlich beides: Beim Vererben erhält die Subklasse auch eventuelle Methodenimplementierungen der Basisklasse. Um zu verhindern, dass man von zwei Klassen zwei verschiedene Implementierungen für ein und dieselbe Methode erbt, verbietet TypeScript – wie viele andere moderne Sprachen auch – Mehrfachvererbung. Das bedeutet, dass eine Klasse nur von einer einzigen anderen Klasse erben kann.

Beim Implementieren stellt der Compiler jedoch nur sicher, dass die Subklasse sämtliche Methoden der Basisklasse (oder auch des im-

plementierten Interface) aufweist. Das stellt sicher, dass sich die Subklasse als Ersatz für die Basisklasse verwenden lässt. Eine Klasse kann beliebig viele andere Klassen und Interfaces implementieren: Da hier keine Implementierungen vererbt werden, können diese auch nicht zueinander in Konflikt stehen.

So gesehen, ist `implements` »unaufdringlicher« und sollte wenn möglich zum Einsatz kommen. Ähnlich wie Interfaces verwendet der TypeScript-Compiler `implements`-Beziehungen nur beim Kompilieren. Sie finden sich aber nicht im Bundle wieder, zumal es in JavaScript dafür keine Entsprechung gibt. Deswegen kommt es beim Einsatz von `implements` nicht so einfach zu wechselseitigen (zyklischen) Verweisen, die spätestens zur Laufzeit zu Problemen führen.

Auch der `DummyFlightService` implementiert die abstrakte Basisklasse `FlightService` (siehe Beispiel 5-6).

*Beispiel 5-6: Der `DummyFlightService` ist eine weitere `FlightService`-Implementierung*

```
// src/app/dummy-flight.service.ts

import { Injectable } from '@angular/core';
import { Observable, of } from 'rxjs';
import { Flight } from './flight';
import { FlightService } from './flight.service';

@Injectable({
  providedIn: 'root'
})
export class DummyFlightService implements FlightService {

  constructor() { }

  find(from: string, to: string): Observable<Flight[]> {
    return of([
      { id: 1, from: 'Frankfurt', to: 'Flagranti', date: '2022-01-02T19:00+01:00' },
      { id: 2, from: 'Frankfurt', to: 'Kognito', date: '2022-01-02T19:30+01:00' },
      { id: 3, from: 'Frankfurt', to: 'Mallorca', date: '2022-01-02T20:00+01:00' }
    ]);
  }
}
```

Die hier verwendete Ausprägung der Methode `find` liefert lediglich ein paar hartcodierte Testflüge zurück. Da die abstrakte Basisklasse als Rückgabewert ein `Observable<Flight[]>` vorgibt, werden diese Flüge mit der Funktion `of` in einem solchen Observable verpackt.

Nun ist es an der Zeit, unsere Änderungen auszuprobieren: Wenn Sie die Lösung starten (`ng serve -o`), sollten Sie zunächst keine Änderung bemerken. Der Grund dafür ist, dass auch der neue `DefaultFlightService` die Flüge von unserer Web-API abrupt.

Vergewissern Sie sich nun, dass dieser Service tatsächlich austauschbar ist. Lassen Sie dazu den `FlightService` auf den `DummyFlightService` verweisen (siehe Beispiel 5-7).

Beispiel 5-7: Der FlightService verweist nun auf den DummyFlightService.

```
// src/app/flight.service.ts

import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import { DummyFlightService } from './dummy-flight.service';
import { Flight } from './flight';

@Injectable({
  providedIn: 'root',
  // Zum Testen auf den DummyFlightService umleiten:
  useClass: DummyFlightService
})
export abstract class FlightService {

  abstract find(from: string, to: string): Observable<Flight[]>

}
```

Jetzt sollten Sie die hartcodierten Flüge des DummyFlightService sehen (siehe Abbildung 5-4).

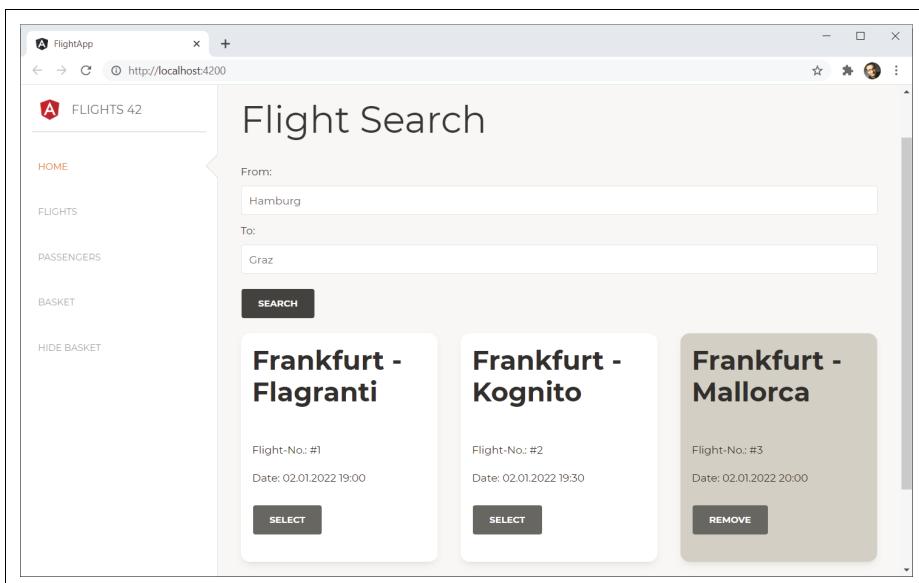


Abbildung 5-4: Das Ergebnis des DummyFlightService



Falls Sie schon Erfahrung mit typisierten Sprachen wie Java oder C# haben, wird Ihnen die hier gezeigte Nutzung von Basistypen sympathisch erscheinen. Möchten Sie jedoch den Service lediglich im Rahmen von automatisierten Tests austauschen, ist es üblich, sich diesen Umweg zu ersparen. Stattdessen können Sie einfach, wie eingangs gezeigt, einen konkreten Service ohne Weiterleitung verwenden:

```

@Injectable({
  providedIn: 'root'
})
export class FlightService {
  [...]
}

```

Innerhalb der automatisierten Tests können Sie auch ohne Verwendung eines gemeinsamen Basistyps diesen Service durch einen anderen austauschen. Das liegt vor allem daran, dass TypeScript in dynamisch typisiertes JavaScript übersetzt wird.

Vergessen Sie nach diesem Experiment nicht, die `FlightService`-Konfiguration rückgängig zu machen, sodass wieder der `DefaultFlightService` zum Einsatz kommt.

Sie haben nun die wichtigsten Aspekte von Service und Dependency Injection kennengelernt. Wer erst mal genug von Services hat, kann gern mit dem nächsten Kapitel fortfahren und bei Bedarf die restlichen Abschnitte später lesen.

Die folgenden Kapitel bauen übrigens auf dem aktuellen Stand auf. Sie sollten so mit Ihr Projekt an dieser Stelle sichern oder die Beispiele der nachfolgenden Abschnitte in einem separaten Branch Ihrer Quellcodeverwaltung ausprobieren.

## Services mit klassischen Providern konfigurieren

In den vorangegangenen Abschnitten haben wir die Services über ihren `Injectable`-Dekorator konfiguriert:

```

@Injectable({
  providedIn: 'root',
  useClass: DefaultFlightService
})
export abstract class FlightService {
  [...]
}

```

Solche Konfigurationen nennt Angular auch Tree-Shakable Provider. Angular bietet außerdem eine alternative Konfigurationsmöglichkeit, bei der die Provider in den Angular-Modulen verstaut werden. Da es sich hierbei um eine ältere Schreibweise handelt, sprechen wir von »klassischen« Providern.

Bei dieser klassischen Variante erhält der Dekorator `Injectable` keine Metadaten – weder die der Basisklasse noch jene der konkreten Implementierungen (siehe Beispiel 5-8).

*Beispiel 5-8: Services ohne Metadaten in `Injectable`*

```
// src/app/flight.service.ts
```

```

@Injectable()
export abstract class FlightService {
  abstract find(from: string, to: string): Observable<Flight[]>;
}

```

```
// src/app/default-flight.service.ts

@Injectable()
export class DefaultFlightService implements FlightService {
    [...]
}
```

```
// src/app/dummy-flight.service.ts

@Injectable()
export class DummyFlightService implements FlightService {
    [...]
}
```

Stattdessen platzieren Sie die Servicekonfiguration in einem Angular-Modul (siehe Beispiel 5-9).

*Beispiel 5-9: AppModule mit Servicekonfiguration (Serviceprovider)*

```
// src/app/app.module.ts

[...]
// Services importieren:
import { FlightService } from './flight.service';
import { DefaultFlightService } from './default-flight.service';

@NgModule({
    imports: [
        [...]
    ],
    declarations: [
        [...]
    ],
    providers: [
        // Serviceprovider hinzufügen:
        {
            provide: FlightService,
            useClass: DefaultFlightService
        }
    ],
    bootstrap: [
        AppComponent
    ]
})
export class AppModule { }
```

Das hier gezeigte Beispiel definiert, dass Komponenten, die einen FlightService anfordern, eine Instanz des DefaultFlightService erhalten.



Auch wenn diese Konfiguration innerhalb eines Moduls erfolgt, gilt sie für die *gesamte* Anwendung. Das hier gezeigte Beispiel entspricht somit dem im vorherigen Abschnitt präsentierten Beispiel, das explizit den Root-Scope im Dekorator Injectable festgelegt hat (`providedIn:`

*root*). Das liegt daran, dass sich Module, die die Anwendung nicht per Lazy Loading beziehen, den Root-Scope teilen.

Lazy Module haben hingegen ihren eigenen Scope. Details dazu finden Sie in Kapitel 13.

Falls Sie für einen Service keine Umleitung konfigurieren wollen, können Sie auch den Service auf sich selbst abbilden:

```
providers: [
  {
    provide: FlightService,
    useClass: FlightService
  }
],
```

Diese Selbstabbildung ist notwendig, um den Service bei Angular zu registrieren. Dieses Beispiel setzt natürlich voraus, dass der `FlightService` wie am Anfang dieses Kapitels eine konkrete Klasse ist. Um zu verhindern, dass Sie hier den Namen des Service doppelt platzieren müssen, existiert die folgende Kurzschreibweise:

```
providers: [
  FlightService,
],
```

Sie können beide Schreibweisen auch kombinieren:

```
providers: [
  {
    provide: FlightService,
    useClass: FlightService
  },
  OtherService
],
```

`OtherService` ist hier ein fiktiver Service, der der Veranschaulichung dient.

## Einen Service lokal registrieren

Die letzten Abschnitte haben gezeigt, wie sich Services global – also für die gesamte Anwendung – registrieren lassen. Alternativ dazu kann eine Anwendung einen Service auch nur für eine bestimmte Komponente registrieren. Solche Services können von dieser Komponente sowie von deren direkten und indirekten Child-Komponenten – sprich, von allen Komponenten darunter – konsumiert werden. In diesem Bereich gilt der Service auch als Singleton.

Abbildung 5-5 veranschaulicht das: Hier wurde für das Token `FlightService` dreimal ein Service registriert: einmal global und zweimal lokal auf Komponentenebene. Somit existieren auch drei Instanzen des `FlightService`.

Generell gilt, dass ein Service für den Teilbaum gültig ist, für den er registriert wurde. Registrieren weiter unten gelegene Komponenten erneut einen Provider für dasselbe Token, überschatten diese den weiter oben eingeführten Service. Aus die-

sem Grund erhält die FlightSearchComponent auch ihren eigenen Service und nicht den Service, der weiter oben an globaler Stelle definiert wurde.

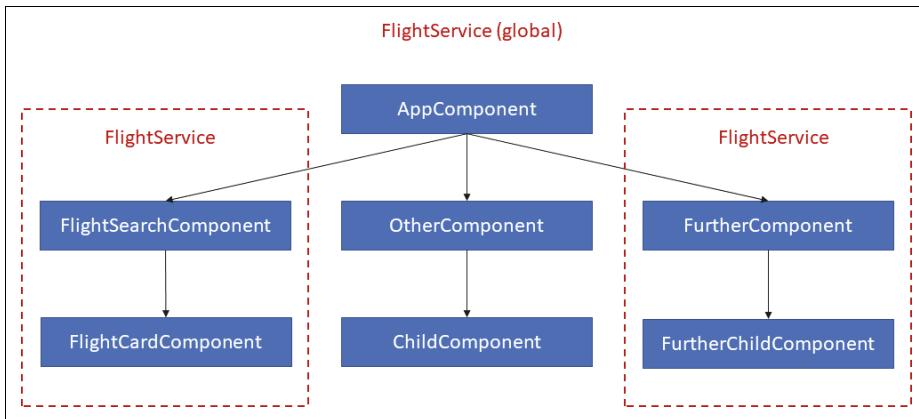


Abbildung 5-5: Services und Scopes

Um Services für eine Komponente zur registrieren, erhält ihr Component-Dekorator entsprechende Provider (siehe Beispiel 5-10).

*Beispiel 5-10: Einen Service auf Komponentenebene deklarieren*

```
// src/app/flight-search/flight-search.component.ts

[...]
import { FlightService } from '../flight.service';
// Diesen Import hinzufügen:
import { DummyFlightService } from '../dummy-flight.service';

@Component({
  [...]
  providers: [
    {
      provide: FlightService,
      useClass: DummyFlightService
    }
  ]
})
export class FlightSearchComponent implements OnInit {

  [...]

  constructor(private flightService: FlightService) {
  }

  [...]
}
```

In diesem Beispiel erhält die FlightSearchComponent den DummyFlightService, selbst wenn auf globaler Ebene der DefaultFlightService konfiguriert ist. Auch sämtliche

direkte und indirekte Child-Komponenten würden den `DummyFlightService` erhalten, sofern sie keinen anderen Provider für den `FlightService` einrichten.

Tree-Shakable Provider erlauben solche Szenarien leider nicht. Sie müssen lokale Services also immer mit klassischen Providern definieren. Allerdings lässt sich diese Schreibweise abkürzen, wenn dieselbe Typ sowohl als Token als auch als Service fungieren soll:

```
@Component({
  [...]
  providers: [
    FlightService
  ]
})
export class FlightSearchComponent implements OnInit {
  [...]
  constructor(private flightService: FlightService) {
  }
  [...]
}
```

Auch dieses Beispiel geht davon aus, dass der `FlightService` wie eingangs gezeigt eine konkrete Klasse ist.

Lokale Services werden übrigens gemeinsam mit ihren Komponenten zerstört. Entledigt sich beispielsweise ein nicht mehr erfülltes `*ngIf` von unserer `FlightSearch` Component, wird auch ihr `FlightService` aus dem Weg geräumt. Eventuelle Zustände, die in diesem Service hinterlegt sind, gehen verloren.

Möchten Sie im Zuge dessen Aufräumarbeiten stattfinden lassen, können Sie diese im Life-Cycle-Hook `ngOnDestroy` platzieren:

```
@Injectable([...])
export class DefaultFlightService implements FlightService, OnDestroy {
  [...]
  ngOnDestroy(): void {
    console.debug('Sag beim Abschied leise Servus!');
  }
  [...]
}
```



Der Life-Cycle-Hook `ngOnDestroy` ist übrigens der einzige, den Angular für Services unterstützt. Alle anderen dürfen Sie nur in Komponenten verwenden.

## Arten von Providern

Bis jetzt haben wir lediglich Provider betrachtet, die ein Token mit klassenbasierten Services verknüpfen. Auch wenn das der wohl häufigste Fall sein mag, möchten wir in diesem Abschnitt weitere Optionen aufzeigen.

## useClass

Hierbei handelt es sich um die bis jetzt verwendete Option: useClass lässt ein Token auf eine Klasse verweisen. Diese wird bei Bedarf als Singleton instanziert und in die jeweiligen Konsumenten injiziert.

Beim Einsatz von Tree-Shakable Providern findet sich useClass direkt im Injectable-Dekorator des Tokens:

```
@Injectable({
  providedIn: 'root',
  useClass: DefaultFlightService
})
export abstract class FlightService { [...] }
```

Bei klassischen Providern gestaltet sich die Verwendung von useClass wie folgt:

```
providers: [
  {
    provide: FlightService,
    useClass: FlightService
  }
],
```

Der Vollständigkeit halber möchten wir hier noch mal auf die Kurzschreibweise hinweisen für die Fälle, in denen derselbe Typ als Token sowie als Service fungiert:

```
providers: [
  FlightService
],
```

## useValue

Mit useValue bindet ein Provider ein Token an einen bereits existierenden Wert. Dabei kann es sich um einen einfachen Wert, z.B. einen String, aber auch um ein Objekt oder eine Funktion handeln.

Beispiel 5-11 definiert zum Beispiel ein Objekt dummyFlightService mit einer find-Methode.

*Beispiel 5-11: Ein Dummy-Objekt als Ersatz für den FlightService*

```
// src/app/flight-service-object.ts

import { of } from 'rxjs';

export const flightServiceObject = {
  find: (from: string, to: string) => {
    console.debug('find', from, to);
    return of([
      { id: 1, from, to, date: new Date().toISOString() }
    ]);
  }
};
```

Die Methode `find` gibt zur Veranschaulichung eine Information auf der Konsole aus und liefert anschließend ein Observable mit einem einzigen Flug zurück. Solche Services bieten sich für Unit-Tests an: Sie lassen sich einfach pro Testmethode definieren und müssen auch nur jene Methoden bzw. Eigenschaften aufweisen, die man im jeweiligen Test benötigt.

Zum Registrieren solcher Objekte bieten Tree-Shakable Provider eine Eigenschaft `useValue` an (siehe Beispiel 5-12).

*Beispiel 5-12: Ein Objekt mit einem Tree-Shakable Provider an ein Token binden*

```
// src/app/flight.service.ts

[...]

// Neuer Import:
import { flightServiceObject } from './flight-service-object';

@Injectable({
  providedIn: 'root',
  // useValue verweist auf das Objekt:
  useValue: flightServiceObject
})
export abstract class FlightService {
  abstract find(from: string, to: string): Observable<Flight[]>;
}
```

In diesem Fall erhält jeder Konsument, der einen `FlightService` via Dependency Injection anfordert, unser Objekt.

Die gleiche Möglichkeit wird auch von klassischen Providern angeboten. Beispiel 5-13 zeigt ein Beispiel dafür.

*Beispiel 5-13: Ein Objekt in einem Angular-Modul an ein Token binden*

```
// src/app/app.module.ts

// Neuer Import:
import { flightServiceObject } from './flight-service-object';

@NgModule({
  imports: [
    [...]
  ],
  declarations: [
    [...]
  ],
  providers: [
    // Provider:
    {
      provide: FlightService,
      useValue: flightServiceObject
    }
  ],
  bootstrap: [
    ...
  ]
})
```

```
        AppComponent  
    ]  
})  
export class AppModule { }
```

## useFactory

Mit `useFactory` bindet ein Provider ein Token an eine Factory. Dabei handelt es sich um eine Funktion, deren Aufgabe die Erzeugung des gewünschten Service ist. Diese Funktion kann sich andere Services injizieren lassen, um ihre Aufgabe zu erledigen.

Da Services Singletons sind, ruft Angular diese Funktion nur ein einziges Mal auf. Zur Demonstration dient hier eine einfache Factory, die entweder einen `DefaultFlightService` oder einen `DummyFlightService` liefert (siehe Beispiel 5-14).

*Beispiel 5-14: Factory für FlightServices*

```
// src/app/flight-service.factory.ts  
  
import { HttpClient } from '@angular/common/http';  
import { DefaultFlightService } from './default-flight.service';  
import { DummyFlightService } from './dummy-flight.service';  
  
const DEBUG = false;  
  
export const createFlightService = (http: HttpClient) => {  
    if (!DEBUG) {  
        return new DefaultFlightService(http);  
    }  
    else {  
        return new DummyFlightService();  
    }  
};
```

Die Factory liegt hier als Lambda-Ausdruck vor. Bitte beachten Sie, dass sie sich den `HttpClient` injizieren lässt, um damit einen `DefaultFlightService` zu erzeugen.

Die Factory lässt sich nun mit der Eigenschaft eines Tree-Shakable Providers registrieren (siehe Beispiel 5-15).

*Beispiel 5-15: Factory mit Tree-Shakable Provider verwenden*

```
// src/app/flight.service.ts  
  
[...]  
// Neuer Import:  
import { createFlightService } from './flight-service.factory';  
  
@Injectable({  
    providedIn: 'root',  
    useFactory: createFlightService,  
    deps: [HttpClient]  
})  
export abstract class FlightService {
```

```
abstract find(from: string, to: string): Observable<Flight[]>;  
}
```

Beachten Sie hier die Eigenschaft `deps`. Sie verweist auf jene Services, die in die Factory injiziert werden sollen. Die Reihenfolge der Einträge muss der Parameterliste der Factory entsprechen. Diese Eigenschaft ist notwendig, weil der Angular-Compiler die Signatur von Funktionen nicht analysieren kann.

Auch `useFactory` wird von klassischen Providern unterstützt (siehe Beispiel 5-16).

*Beispiel 5-16: Factory mit Provider in Angular-Modul verwenden*

```
// src/app/app.module.ts  
  
[...]  
  
// Neuer Import:  
import { createFlightService } from './flight-service.factory';  
  
@NgModule({  
    imports: [  
        [...]  
    ],  
    declarations: [  
        [...]  
    ],  
    providers: [  
        {  
            provide: FlightService,  
            useFactory: createFlightService,  
            deps: [HttpClient]  
        },  
        [...]  
    ],  
    bootstrap: [  
        AppComponent  
    ]  
})  
export class AppModule {}
```

## useExisting

Die Eigenschaft `useExisting` erlaubt es Providern, eine Weiterleitung auf ein anderes Token einzurichten. Am besten lässt sich diese Art von Provider mit zwei klassischen Providern veranschaulichen:

```
providers: [  
    { provide: FlightService, useExisting: DummyFlightService }  
    { provide: DummyFlightService, useClass: OtherDummyFlightService },  
],
```

Das Token `FlightService` leitet hier auf das Token `DummyFlightService` weiter. Letzteres verweist wiederum auf den hier nicht abgebildeten `OtherDummyFlightService`. Fordert nun ein Konsument einen `FlightService` an, liefert Angular eine Instanz von `OtherDummyFlightService`.

Ein Gegenstück zu diesem Beispiel, das Tree-Shakable Provider nutzt, findet sich in Beispiel 5-17.

*Beispiel 5-17: Tree-Shakable Provider mit useExisting*

```
// src/app/flight.service.ts

[...]
import { DummyFlightService } from './dummy-flight.service';

@Injectable(
{
  providedIn: 'root',
  // Verweis auf weiteres Token
  useExisting: DummyFlightService
}
)
export abstract class FlightService {
  abstract find(from: string, to: string): Observable<Flight[]>;
}
```

Der `DummyFlightService` würde in diesem Fall mit seinem Tree-Shakable Provider und `useClass` auf den `OtherDummyFlightService` weiterleiten.

Umleitungen sind nützlich, wenn Komponenten verschiedene Tokens für ein und dasselbe Konzept verwenden. Natürlich könnten Sie stattdessen auch mehrere separate Provider einrichten:

```
providers: [
  { provide: FlightService, useClass: OtherDummyFlightService },
  { provide: DummyFlightService, useClass: OtherDummyFlightService },
],
```

In diesem Fall bekämen Sie jedoch jeweils einen Singleton für `FlightService_` und für `DummyFlightService`.

Zugegeben, die Fälle, in denen `useExisting` benötigt wird, sind äußerst selten. Kapitel 9 zeigt so einen Fall im Kontext der Formularvalidierung.

## multi

Um besonders flexible Teilsysteme zu kreieren, muss man ab und zu mehrere Services an ein Token binden. Ein Beispiel dafür ist ein Plug-in-System, bei dem eine Anwendung mit mehreren `FlightServices` gleichzeitig arbeitet, um bei verschiedenen Anbietern nach Flügen zu suchen. Idealerweise registriert man die einzelnen `FlightServices` in einem zentralen Modul, und die einzelnen Konsumenten erhalten Zugriff auf sie in Form eines Arrays. Die Konsumenten können dann das Array iterieren und parallel bei mehreren Anbietern nach Flügen suchen.

Das Schöne an diesem Ansatz ist die Flexibilität: Soll die Anwendung irgendwann einmal weitere Anbieter unterstützen, muss sie nur einen weiteren `FlightService` registrieren. Die einzelnen Konsumenten bleiben davon unberührt: Sie iterieren nach wie vor alle vorhandenen Services und nutzen sie.

Zur Demonstration registriert Beispiel 5-18 zwei FlightServices.

*Beispiel 5-18: Registrieren von Multi-Providern*

```
[...]
providers: [
  {
    provide: FlightService,
    useClass: DefaultFlightService,
    multi: true
  },
  {
    provide: FlightService,
    useClass: DummyFlightService,
    multi: true
  }
],
[...]
```

Da beide Provider dasselbe Token verwenden, würden sie sich normalerweise gegenseitig überschreiben. Um das zu verhindern, kommt die Eigenschaft `multi` zum Einsatz. Sie legt fest, dass Angular alle registrierten Provider berücksichtigen soll. Als Konsequenz erhalten die Konsumenten ein Array mit `FlightServices` (siehe Beispiel 5-19).

*Beispiel 5-19: Injektion mit Multi-Provider*

```
// src/app/flight-search/flight-search.component.ts

// Inject importieren:
import { Component, Inject, OnInit } from '@angular/core';
[...]

@Component( [...] )
export class FlightSearchComponent implements OnInit {

  [...]

  constructor(@Inject(FlightService) private flightServices: FlightService[]) {
  }

  search(): void {
    [...]
  }

  [...]
}
```

Hier ergibt sich leider eine kleine Herausforderung: Das Token ist nach wie vor `FlightService`. Durch die Multi-Provider erhalten wir allerdings ein `FlightService`-Array (`FlightService[]`). Der Datentyp entspricht also nicht mehr dem Token. Aus diesem Grund nutzt das gezeigte Beispiel den Dekorator `Inject`, der auf das zu nutzende Token verweist.



Arrays lassen sich leider nicht als Token verwenden. Deswegen ist das Token hier nach wie vor FlightService. Eine häufig verwendete Alternative dazu bieten die weiter unten beschriebenen konstantenbasierten Tokens.

Nun müssen wir nur noch das FlightService-Array zum Suchen nach Flügen einsetzen (siehe Beispiel 5-20).

*Beispiel 5-20: Mehrere FlightService-Instanzen verwenden*

```
// src/app/flight-search/flight-search.component.ts

[...]

// Neuer Import:
import { merge } from 'rxjs';

@Component( [...] )
export class FlightSearchComponent implements OnInit {

  constructor(@Inject(FlightService) private flightServices: FlightService[]) {
  }

  ngOnInit(): void {
  }

  search(): void {
    this.flights = [];

    const observables = this.flightServices.map(fs => fs.find(this.from, this.to));

    // Ergebnisse der einzelnen Observables mergen
    const observable = merge(...observables);
    // Entspricht merge(observables[0], observables[1], ...)

    observable.subscribe({
      next: (additionalFlights) => {
        this.flights = [...this.flights, ...additionalFlights];
        // Entspricht [this.flights[0], this.flights[1], ...,
        //   additionalFlights[0], additionalFlights[1], ...]
      },
      error: (err) => {
        console.debug('Error', err);
      }
    });
  }
}

[...]
}
```

Zugegeben, diese Implementierung von search ist nicht ganz einfach. Schauen wir uns die einzelnen Schritte etwas genauer an:

- Die Methode startet mit einem leeren flights-Array, das nach und nach mit den Ergebnissen der einzelnen FlightServices erweitert wird.
- Die Array-Methode map bildet die erhaltenen FlightServices auf das Ergebnis ihrer find-Methoden ab. Aus [flightService1, flightService2, ...] wird also [observableMitErgebnis1, observableMitErgebnis2, ...].
- Da die einzelnen Ergebnisse Observables sind, fügt merge sie zu einem einzigen Observable, das die einzelnen Ergebnisse nach und nach veröffentlicht, zusammen. Der Spread-Operator (drei Punkte) bewirkt, dass jeder Eintrag als separater Parameter übergeben wird.
- Die mit subscribe registrierte next-Methode nimmt nach und nach die einzelnen Ergebnisse entgegen und fügt sie am Ende des Arrays flights ein.



Multi-Provider lassen sich übrigens nur mit klassischen Providern einrichten. Tree-Shakable Provider können hierfür leider nicht verwendet werden.

## Konstanten als Tokens

Nicht für jedes Konzept findet sich ein Basistyp, den man auch als Token verwenden kann. Beispiele dafür sind Services, die durch einfache Werte (wie Strings), Arrays (die sich nicht als Token verwenden lassen) oder auch durch Funktionen repräsentiert werden. Für diese Fälle bietet Angular die Möglichkeit, Konstanten als Tokens einzusetzen.

Für solche Konstanten definiert Angular den Typ `InjectionToken` (siehe Beispiel 5-21).

*Beispiel 5-21: Konstante als Token*

```
// src/app/tokens.ts

import { InjectionToken } from '@angular/core';

export const BASE_URL = new InjectionToken<string>('BASE_URL', {
  providedIn: 'root',
  factory: () => 'http://demo.ANGLARarchitects.io/api/'
});
```

Dieses Beispiel definiert ein `InjectionToken`, das die Basis-URL unserer Web-API veröffentlicht. Auch wenn diese URL lediglich ein String ist, handelt es sich dabei technisch gesehen um einen Service.

Der Typparameter definiert den Typ des Service. Der erste übergebene Parameter ist lediglich ein String, der beim Debuggen angezeigt wird. Typischerweise ent-

spricht er dem Namen der Konstanten. Das hilft, Fehlermeldungen besser zuordnen zu können.

Der zweite Parameter ist ein Objekt mit einem Tree-Shakable Provider. Im Gegensatz zu den bisher betrachteten Tree-Shakable Providern nehmen konstantenbasierte Provider nur eine Factory entgegen.

Der Konsument kann nun diesen Service unter Verwendung des Inject-Dekorators anfordern (siehe Beispiel 5-22).

*Beispiel 5-22: Service mit Konstante anfordern*

```
// src/app/default-flight.service.ts

// Diesen Import ergänzen:
import { BASE_URL } from './tokens';

// Inject importieren:
import { Inject, Injectable } from '@angular/core';
[...]

@Injectable({
  providedIn: 'root'
})
export class DefaultFlightService implements FlightService {

  constructor(
    private http: HttpClient,
    // BASE_URL injizieren lassen:
    @Inject(BASE_URL) private baseUrl: string,
  ) { }

  find(from: string, to: string): Observable<Flight[]> {
    const url = this.baseUrl + 'flight';

    const headers = new HttpHeaders()
      .set('Accept', 'application/json');

    const params = new HttpParams()
      .set('from', from)
      .set('to', to);

    return this.http.get<Flight[]>(url, {headers, params});
  }
}
```

Der Einsatz von `Inject` ist hier notwendig, da der Datentyp (`string`) nicht dem Token (`BASE_URL`) entspricht.

Konstantenbasierte Tokens lassen sich auch für klassische Provider in Modulen oder Komponenten verwenden. In diesem Fall entfällt bei der Instanziierung des Tokens der zweite Parameter:

```
export const BASE_URL = new InjectionToken<string>('BASE_URL');
```

Anschließend kann ein Provider darauf verweisen:

```
providers: [
  {
    provide: BASE_URL,
    useValue: 'http://demo.ANGLULARarchitects.io/api/'
  }
],
```

InjectionTokens kommen häufig für Multi-Provider zum Einsatz, zumal die dahinterstehenden Arrays nicht als Tokens verwendet werden dürfen. Für den im Abschnitt »multi« auf Seite 142 verwendeten Provider könnten Sie beispielsweise das folgende `InjectionToken` definieren:

```
export const FLIGHT_SERVICES = new InjectionToken<FlightService>('FLIGHT_SERVICES');
```

Die darauf basierenden Provider gestalten sich wie folgt:

```
providers: [
  {
    provide: FLIGHT_SERVICES,
    useClass: DefaultFlightService,
    multi: true
  },
  {
    provide: FLIGHT_SERVICES,
    useClass: DummyFlightService,
    multi: true
  },
]
```

Beim Injizieren dieser Services wird dieses Token an `Inject` übergeben:

```
[...]
constructor(@Inject(FLIGHT_SERVICES) private flightServices: FlightService[]) {
}
[...]
```

## Zusammenfassung

Services bieten wiederverwendbare Funktionalitäten an und können per Dependency Injection von Komponenten, aber auch von anderen Services bezogen werden. Dazu fordern die Konsumenten den Service über Konstruktorargumente an. Welche konkrete Serviceimplementierung Angular tatsächlich liefert, hängt von der aktuellen Konfiguration ab. So kann Angular beispielsweise für Testfälle andere Implementierungen bereitstellen als für den Produktiveinsatz.

Die Konfiguration von Services erfolgt über Provider. Neben den klassischen Providern, die in Angular-Modulen oder Komponenten eingetragen werden, bietet Angular mittlerweile auch die moderneren Tree-Shakable Provider. Sie nehmen die Konfiguration bei der Deklaration des Tokens entgegen.



# KAPITEL 6

# Pipes

Nicht immer weisen die verwendeten Daten jene Struktur auf, die unsere Komponenten für die Ausgabe benötigen. In solchen Fällen müssen Sie Ihre Objektstrukturen formatieren, sortieren oder filtern. Angular bietet für diese Aufgabe das Konzept der Pipes, mit dem sich Daten im Rahmen der Datenbindung transformieren lassen. Dieses Kapitel zeigt, wie Pipes zu verwenden sind, welche Pipes Angular ab Werk bietet und wie eine Anwendung eigene Pipes ins Spiel bringen kann.

## Überblick

Um mit einer Pipe Daten im Zuge der Datenbindung zu transformieren, kommt jene Schreibweise zum Einsatz, die Sie von der Kommandozeile her kennen:

```
 {{ flight.from | city }}
```

Demzufolge übergibt Angular den Wert auf der linken Seite des Pipe-Symbols | an die Pipe auf der rechten Seite. Konkret leitet Angular im betrachteten Fall also `flight.from` an `city` weiter. `city` ist hier eine benutzerdefinierte Pipe, deren Aufbau weiter unten betrachtet wird.

Eine Pipe kann auch zusätzliche Parameter erhalten. Diese hängen Sie, getrennt durch jeweils einen Doppelpunkt, an den Aufruf an:

```
 {{ flight.from | city:'short' }}  
 {{ flight.from | city:'short':'en' }}
```

Außerdem lässt sich das Ergebnis einer Pipe an eine weitere Pipe übergeben:

```
 {{ flight.from | city:'short' | lowercase }}
```

## Built-in-Pipes

Angular bietet ein paar Pipes ab Werk. Eine Übersicht sehen Sie in Tabelle 6-1.

Tabelle 6-1: Built-in-Pipes

Pipe	Beschreibung	Beispiel
date	Formatiert ein Datum entsprechend der übergebenen Formatierungszeichenfolge. Kann mit einem ISO-String oder einem Date-Objekt verwendet werden.	<code>{{ flight.date   date:'dd.MM.yyyy' }}</code>
json	Wandelt ein Objekt in einen JSON-String um. Das kann während der Entwicklung für Testausgaben nützlich sein.	<code>{{ flight   json }}</code>
uppercase	Wandelt einen String in Großbuchstaben um.	<code>{{ flight.from   uppercase }}</code>
lowercase	Wandelt einen String in Kleinbuchstaben um.	<code>{{ flight.from   lowercase }}</code>
titlecase	Wandelt einen String in Titlecase um. Das bedeutet, dass der erste Buchstabe jedes Worts großgeschrieben wird.	<code>{{ flight.from   titlecase }}</code>
number	Formatiert eine Zahl. Erhält als Parameter einen String im Format 'a.b-c', wobei <i>a</i> für die Mindestanzahl an Stellen vor dem Komma, <i>b</i> für die Mindestanzahl an Stellen nach dem Komma und <i>c</i> für die maximale Anzahl an Stellen nach dem Komma steht.	<code>{{ price   number:'1.2-2' }}</code>
currency	Wie number, allerdings wird vor der Zahl eine Währungsbezeichnung eingefügt. Der erste Parameter repräsentiert die Währung in Form eines ISO-Codes (z.B. EUR oder USD), der zweite legt fest, ob anstatt des ISO-Codes das Währungssymbol anzusegn ist (z.B. ? oder \$), und der dritte Parameter entspricht der Formationsangabe der Pipe number.	<code>{{ price   currency:'EUR':false:'1.2-2' }}</code>
percent	Formatiert einen Prozentwert. Aus 0.3 wird beispielsweise 30%. Der optionale Parameter entspricht jenem der Pipe number.	<code>{{ factor   percent:'1.2-2' }}</code>
slice	Filtert ein Array, indem es nur einen bestimmten Indexbereich zurückliefert.	<code>&lt;li *ngFor="let f of flights   slice:1:3"&gt;{{f.from}}&lt;/li&gt;</code>
keyvalue	Liefert Schlüssel und Werte eines Dictionary.	<code>&lt;li *ngFor="let b of basket   keyvalue"&gt;{{b.key}}={{b.value}}&lt;/li&gt;</code>
async	Registriert sich für einen Promise oder ein Observable (subscribe) und liefert den empfangenen Wert zurück. Infos dazu finden Sie in den Kapiteln 11 und 13.	<code>{{ myObservable   async }}</code>



Die Pipes number, currency und date können auch angewiesen werden, lokale Formate zu nutzen. Ein Beispiel dafür ist ein deutsches Datum (z. B. 20.1.1981) oder eine britische Dezimalzahl mit Tausendertrennzeichen (z. B. 1,701.05). Details dazu finden Sie in Kapitel 16.

## Eigene Pipes

Nachdem wir im letzten Abschnitt die Built-in-Pipes vorgestellt haben, zeigt dieser Abschnitt, wie eine Anwendung eigene Pipes definieren kann. Dazu erklären wir

zunächst die Idee der puren (reinen) Pipes und zeigen danach anhand einer Implementierung, wie eine eigene Pipe zu gestalten ist.

## Pure Pipes

Zur Performanceoptimierung hat das Angular-Team den Begriff der puren (reinen) Pipes eingeführt. Im Sinne der funktionalen Programmierung ist eine Pipe dann pur, wenn ihre Ausgaben einzig und allein von den Eingaben abhängen. Ergibt sich beispielsweise beim Aufruf von

```
 {{ flight.from | city:'short'}}
```

das Ergebnis nur aus `flight.from` und aus dem Parameter `short`, ist die Pipe eine pure Pipe. Verwendet die Pipe zum Ermitteln des Ergebnisses jedoch auch andere Informationen, zum Beispiel die aktuelle Uhrzeit oder Ergebnisse eines Serviceaufrufs, ist sie per definitionem nicht mehr pur.

Die Entscheidung zwischen pur und nicht pur macht einen großen Unterschied für die Performance. Pipes, die nicht pur sind, muss Angular nach jedem Event ausführen. Der Grund dafür ist, dass jeder Event-Handler prinzipiell einen Nebeneffekt haben kann, der das Ergebnis der Pipe beeinflusst.

Ein Beispiel dafür wäre ein Service, der die für den Benutzer konfigurierte Sprache verwaltet. Wenn diese Sprache das Ergebnis einer Pipe beeinflusst, kann sich das nach jedem Event ändern, zumal jeder Event-Handler die im Service gespeicherte Sprache verändern kann.

Pipes, die hingegen pur sind, muss Angular zunächst nur einmal ausführen. Danach kann das Framework den erhaltenen Wert wiederverwenden, bis sich eine der Eingaben verändert. Erst dann muss es die Pipe erneut anstoßen.

Ob eine Pipe pur ist oder nicht, legt der Entwickler fest, wenn er sie deklariert. Angular glaubt diesen Angaben und ruft die Pipe dementsprechend oft erneut auf. Eine falsche Angabe wirkt sich somit negativ auf die Performance bzw. auf die Aktualisierung der UI aus.



So gut wie alle Build-in-Pipes sind pur. Die einzige Ausnahme bildet die JSON-Pipe. Sie aktualisiert die Ausgabe, selbst wenn sich einzelne Eigenschaften des gebundenen Objekts ändern.

## Implementierung einer einfachen Pipe

Auch Pipes lassen sich mit der Angular CLI generieren:

```
ng generate pipe city
```

Die Kurzschreibweise dafür lautet

```
ng g p city
```



Haben Sie für Visual Studio Code das Plug-in *Angular Schematics* installiert, können Sie stattdessen den Befehl *Angular: Generate another schematic* im Kontextmenü des jeweiligen Ordners nutzen (siehe Abbildung 6-1). Danach wählen Sie aus einer Liste den Eintrag *Pipe*.

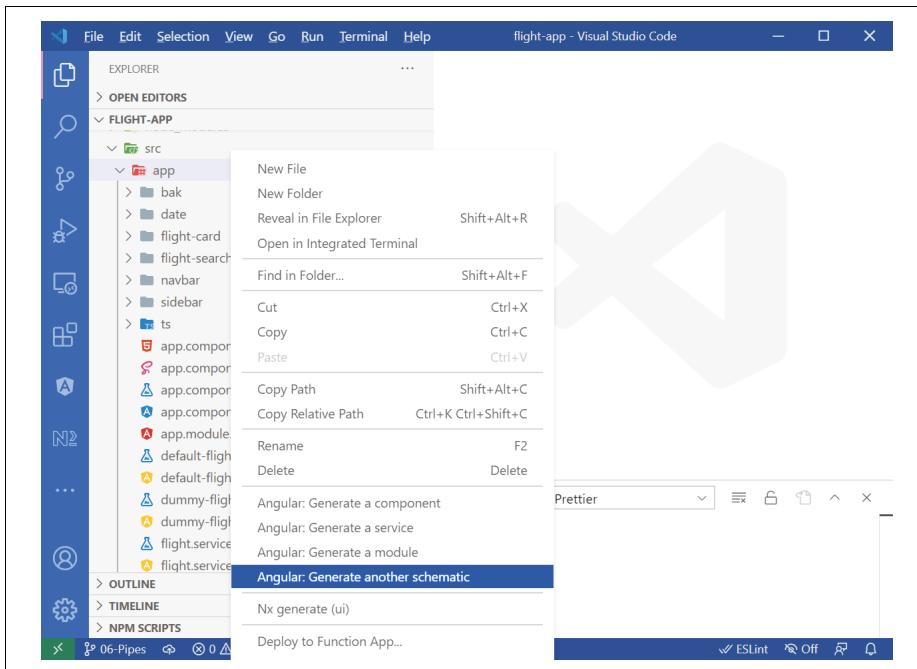


Abbildung 6-1: Pipe mit Visual Studio Code generieren

Das für die Pipe generierte Grundgerüst besteht aus einer Klasse mit einem Pipe-Dekorator (siehe Beispiel 6-1).

#### Beispiel 6-1: Für CityPipe generiertes Grundgerüst

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'city'
})
export class CityPipe implements PipeTransform {

  transform(value: unknown, ...args: unknown[]): unknown {
    return null;
  }
}
```

Der Pipe-Dekorator legt jenen Namen fest, der im HTML zum Aufrufen der Pipe zu nutzen ist. Die Klasse implementiert auch das Interface `PipeTransform`, das die Methode `transform` vorgibt. Diese Methode führt Angular aus, um das Ergebnis der Pipe abzurufen.

Ihr erster Parameter erhält den gepfosten Wert. Alle anderen Parameter entsprechen den an die Pipe angehängten Parametern. Ein Aufruf von

```
{} flight.from | city:'short':'en' {}
```

führt beispielsweise zu folgendem Aufruf der `transform`-Methode:

```
cityPipe.transform(flight.from, 'short', 'en');
```

Eine einfache Implementierung unseres Grundgerüsts findet sich in Beispiel 6-2.

*Beispiel 6-2: Implementierung der CityPipe*

```
// src/app/city.pipe.ts

import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'city',
  pure: true
})
export class CityPipe implements PipeTransform {
  transform(value: string | undefined, format: 'long' | 'short'): string | undefined {

    let short;
    let long;

    switch (value) {
      case 'Graz':
        short = 'GRZ';
        long = 'Flughafen Graz Thalerhof';
        break;
      case 'Hamburg':
        short = 'HAM';
        long = 'Airport Hamburg Fuhlsbüttel Helmut Schmidt';
        break;
      default:
        short = long = value;
        // Formatierung nicht möglich ...
    }

    if (format === 'long') {
      return long;
    }

    return short;
  }
}
```

Die Methode `transform` bildet nun den gepfosten Wert sowie den Parameter `format` in einen offiziellen Flughafennamen um. Zu diesem Zweck haben wir die Datentypen der Parameter und des Rückgabewerts konkretisiert. Als `format` kann der Aufrufer beispielsweise nur mehr die Strings `short` und `long` übergeben.

Außerdem wurde der Pipe-Dekorator um eine Eigenschaft `pure:true` erweitert. Diese Einstellung teilt Angular mit, dass es sich um eine pure Pipe handelt. Somit kann Angular die oben diskutierte Optimierung durchführen.

## Pipes registrieren und nutzen

Bevor wir die Pipe nutzen können, müssen wir uns vergewissern, dass sie bei einem Angular-Modul registriert ist. Nur in diesem Fall kennt Angular die Pipe und kann sie ausführen. Werfen Sie dazu einen Blick in Ihr `AppModule`. Hier sollten Sie unter `declarations` Ihre `CityPipe` wiederfinden (siehe Beispiel 6-3).

*Beispiel 6-3: Die `CityPipe` wurde im `AppModule` registriert.*

```
// src/app/app.module.ts

[...]
// Neuer Import:
import { CityPipe } from './city.pipe';

@NgModule({
  imports: [
    [...]
  ],
  declarations: [
    [...]
    // Neuer Eintrag:
    CityPipe
  ],
  providers: [],
  bootstrap: [
    AppComponent
  ]
})
export class AppModule { }
```

Dieser Eintrag wird beim Generieren des Grundgerüsts der Pipe erzeugt. Finden Sie diesen Eintrag nicht, müssen Sie ihn manuell hinzufügen.

Anschließend können Sie die Pipe in sämtlichen Komponenten dieses Moduls, z.B. in der Datei `flight-card.component.html`, nutzen:

```
<h2 class="title">
  {{item?.from | city:'short' }} - {{item?.to | city:'long' }}
</h2>
```

Führen Sie Ihre Anwendung nun aus (`ng serve -o`), sollte sich unsere Änderung wie in Abbildung 6-2 manifestieren.

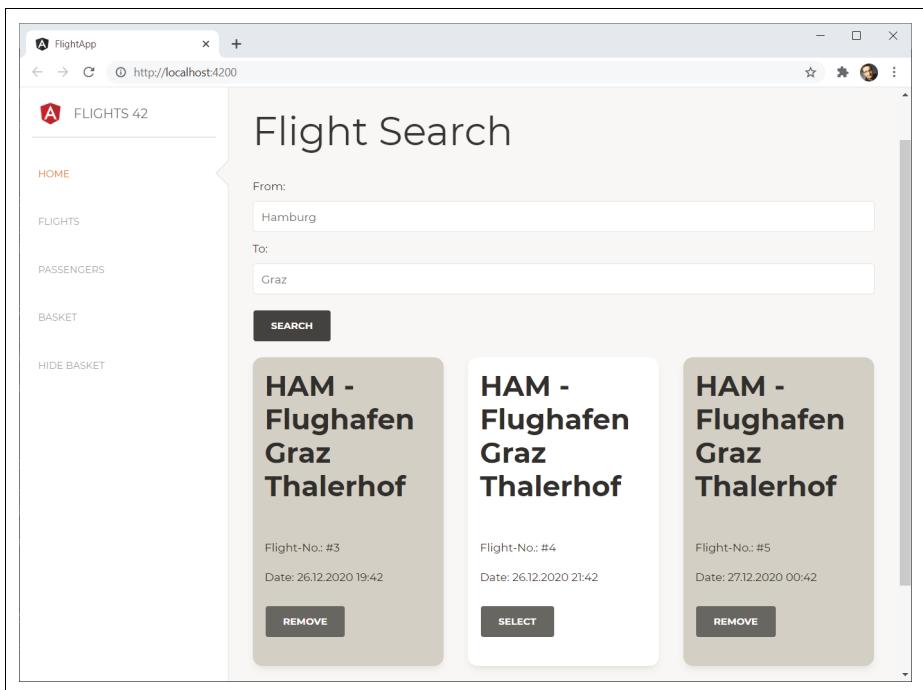


Abbildung 6-2: FlightCards mit formatierten Flughafenbezeichnungen



Pipes lassen sich nur im Rahmen von Property-Bindings verwenden. Für Two-Way-Data-Bindings sind sie hingegen nicht geeignet. Für solche Zwecke existiert jedoch ein Angular-interner Mechanismus namens ControlValueAccessor. Informationen dazu finden Sie unter <https://www.angulararchitects.io/aktuelles/parser-und-formatter-in-angular-2/>.

## Weiterführende Konstellationen

Bis jetzt haben wir die wichtigsten Aspekte von Pipes besprochen. Falls Ihnen das genügt, können Sie gern mit dem nächsten Kapitel weitermachen und bei Bedarf hierher zurückkehren. Möchten Sie dagegen weitere Infos zu Möglichkeiten rund um Pipes bekommen, sind Sie hier genau richtig.

Um Platz zu sparen, erwähnen wir jetzt nicht mehr bei jeder Pipe, dass sie mit `ng generate pipe` erzeugt sowie in einem Angular-Modul deklariert werden muss.

## Pipes und Objekte

Pipes können auch ganze Objekte entgegennehmen und zurückliefern. Die Status FilterPipe in Beispiel 6-4 nimmt beispielsweise ein Flight-Array entgegen und liefert eine gefilterte Version davon zurück.

Beispiel 6-4: Die StatusFilterPipe filtert das übergebene Array.

```
// src/app/status-filter.pipe.ts

import { Pipe, PipeTransform } from '@angular/core';
import { Flight } from './flight';

@Pipe({
  name: 'statusFilter',
  pure: true
})
export class StatusFilterPipe implements PipeTransform {

  transform(flights: Flight[], delayedFilter: boolean | undefined): Flight[] {
    if (typeof delayedFilter === 'undefined') {
      return flights;
    }

    return flights.filter(f => f.delayed === delayedFilter);
  }
}
```

Diese Pipe lässt sich direkt in der `ngFor`-Direktive in der Datei `flight-search.component.html` verwenden:

```
<!-- src/app/flight-search/flight-search.component.html -->

[...]

<div *ngFor="let f of flights | statusFilter:false" [...]>
  <app-flight-card [item]="f" [(selected)]="basket[f.id]">
    </app-flight-card>
</div>

[...]
```

Der an `statusFilter` übergebene Boolean muss natürlich nicht hartcodiert sein, Sie können auch eine Eigenschaft aus der jeweiligen Komponente angeben. Hier verwenden wir zum Beispiel die Eigenschaft `delayFilter`:

```
<!-- src/app/flight-search/flight-search.component.html -->

[...]

<div *ngFor="let f of flights | statusFilter:delayFilter" [...]>
  <flight-card [item]="f" [(selected)]="basket[f.id]">
    </flight-card>
</div>

[...]
```

Die Eigenschaft `delayFilter` wird analog zu den anderen, z.B. `from`, `to` oder `flights`, in der Komponente hinterlegt:

```
// src/app/flight-search/flight-search.component.ts

[...]

@Component({ [...] })
export class FlightSearchComponent implements OnInit {
    [...]
    delayFilter = false;
    [...]
}
```

Zum Aktualisieren solcher Eigenschaften bieten sich Event-Handler oder Two-Way-Data-Bindings an:

```
<!-- src/app/flight-search/flight-search.component.html -->

[...]

<div class="form-group">
    <input [(ngModel)]="delayFilter" type="checkbox" id="delayFilter"
    name="delayFilter">
    &nbsp;
    <label for="delayFilter">Delayed</label>
</div>

[...]
```

## Pipes und Direktiven

Pipes lassen sich auch mit Direktiven kombinieren. Zur Demonstration verwenden wir hier eine StatusColorPipe, die den Status eines Flugs auf eine Farbbezeichnung abbildet (siehe Beispiel 6-5).

*Beispiel 6-5: Die StatusColorPipe liefert abhängig vom übergebenen Status eine Farbbezeichnung zurück.*

```
// src/app/status-color.pipe.ts

import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
    name: 'statusColor',
    pure: true
})
export class StatusColorPipe implements PipeTransform {

    transform(delayed: boolean | undefined): string {
        if (delayed) {
            return 'darkred';
        }
        else {
            return 'darkgreen';
        }
    }
}
```

Diese Pipe lässt sich direkt innerhalb einer `ngStyle`-Direktive aufrufen:

```
<!-- src/app/flight-card/flight-card.component.html -->
[...]
<div class="card-body">
  [...]
  <p [ngStyle]="{ 'color': item?.delayed | statusColor }">Delayed:
  {{ item?.delayed }}</p>
  [...]
</div>
```

Somit erhalten wir ein Styling, das den aktuellen Zustand widerspiegelt. Um das Hartcodieren von Farbwerten und -bezeichnern in Direktiven zu vermeiden, könnten Sie auch die Namen von Klassen zurückliefern. Diese Klassen können per (S)CSS gestylt und mit der in Kapitel 3 besprochenen `ngClass`-Direktive zugewiesen werden.

## Pipes und Services

Gerade größere Logiken möchte man in Pipes vermeiden und stattdessen in Services auslagern. Diese Services lassen sich per Dependency Injection in der Pipe anfordern. Lassen Sie uns beispielsweise annehmen, dass wir die Logik der `CityPipe` in einen `CityService` ausgelagert haben (siehe Beispiel 6-6).

*Beispiel 6-6: Der CityService formatiert die Namen von Flughäfen.*

```
// src/app/city.service.ts

import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class CityService {

  constructor() { }

  formatName(value: string, format: 'short' | 'long'): string {
    let short;
    let long;

    switch (value) {
      case 'Graz':
        short = 'GRZ';
        long = 'Flughafen Graz Thalerhof';
        break;
      case 'Hamburg':
        short = 'HAM';
        long = 'Airport Hamburg Fuhlsbüttel Helmut Schmidt';
        break;
      default:
        short = long = value;
    }
  }
}
```

```

        // Formatierung nicht möglich ...
    }

    if (format === 'long') {
        return long;
    }

    return short;
}
}

```

Die CityPipe kann nun den neu geschaffenen CityService mit einem Konstruktor-argument anfordern und in seiner transform-Methode nutzen (siehe Beispiel 6-7).

*Beispiel 6-7: Die CityPipe lässt sich den CityService injizieren.*

```
// src/app/city.pipe.ts
```

```

import { Pipe, PipeTransform } from '@angular/core';
import { CityService } from './city.service';

@Pipe({
    name: 'city',
    pure: true
})
export class CityPipe implements PipeTransform {

    constructor(private cityService: CityService) {}

    transform(value: string | undefined, format: 'long' | 'short'): string | undefined {
        if (typeof value === 'undefined') {
            return value;
        }

        return this.cityService.formatName(value, format);
    }
}

```



Auch wenn es technisch möglich ist, sollten Sie HTTP-Aufrufe innerhalb von transform vermeiden. Das führt zum einen zu sehr viel Netzwerkverkehr und wirkt sich schlecht auf die Performance aus. Außerdem können Sie mit einem solchen Vorgehen sehr schnell Endlosschleifen provozieren. Auf jeden Fall würden wir solch ein Konstrukt als Code-Smell ansehen und es bei einer Review sehr genau hinterfragen.

## Aufräumarbeiten mit `ngOnDestroy`

Es kommt nicht häufig vor, aber ab und an müssen Sie innerhalb von Pipes Aufräumarbeiten erledigen lassen. Hierzu können Sie den Life-Cycle-Hook `ngOnDestroy` implementieren (siehe Beispiel 6-8).

*Beispiel 6-8: Pipes können den Life-Cycle-Hook OnDestroy implementieren.*

```
// src/app/city.pipe.ts  
[...]  
  
@Pipe([...])  
export class CityPipe implements PipeTransform, OnDestroy {  
    [...]  
  
    ngOnDestroy(): void {  
        console.debug('Ich nehme den Hut und sage Adieu!');  
    }  
  
    [...]  
}
```

Wie bei Services ist `ngOnDestroy` auch bei Pipes der einzige unterstützte Life-Cycle-Hook. Alle anderen von Angular angebotenen Hooks funktionieren nur mit Komponenten.

## Zusammenfassung

Pipes erlauben das Transformieren von Daten im Zuge der Datenbindung. Somit kann eine Anwendung die vom Server empfangenen Daten weiterverwenden, ohne sie in eigene Strukturen überführen zu müssen. Stattdessen bereitet sie die Daten mit einer Pipe für den jeweiligen Anwendungsfall auf.

Neben ein paar Built-in-Pipes, die sich im Lieferumfang von Angular befinden, können Sie auch eigene Pipes schreiben. Sind diese Pipes lediglich von Ihren Eingaben abhängig, ist von puren Pipes die Rede. Angular kann deren Ausführung optimieren.

# KAPITEL 7

# Module

Angular verwendet Module, um zusammengehörige Codeeinheiten zusammenzufassen. In diesem Kapitel stellen wir eine typische Modulstruktur vor, die Angular-Anwendungen verwenden, und zeigen, wie sich unterschiedliche Arten von Modulen einrichten lassen.

## Motivation

In den bisher betrachteten Abschnitten haben wir uns mit einem einzigen Modul zufriedengegeben. Dabei handelte es sich um das `AppModule`, das die Anwendung beim Bootstrapping an Angular übergibt. Wie Beispiel 7-1 zeigt, deckt dieses Modul drei verschiedene Aspekte ab: Zum einen beinhaltet es Komponenten, die die Shell der Anwendung darstellt. Damit meinen wir die Komponenten, die die visuelle Grundstruktur der Anwendung implementieren: die `AppComponent` sowie die `SidebarComponent` und `NavbarComponent`.

Zum anderen enthält unser `AppModule` mit der `FlightSearchComponent` und der `FlightCardComponent` zwei Komponenten, die zum Buchen von Flügen eingesetzt werden. Hier geht es also um einen bestimmten Anwendungsfall.

Dann sind da noch die `DateComponent` und die Pipes, die in mehreren Anwendungsfällen zum Einsatz kommen können.

*Beispiel 7-1: Unser unübersichtliches AppModule*

```
// src/app/app.module.ts
```

```
[...]
```

```
@NgModule({
  imports: [
    FormsModule,
    HttpClientModule,
    BrowserModule
  ],
  declarations: [
```

```

// Shell
AppComponent,
SidebarComponent,
NavbarComponent,

// Flugbuchung
FlightSearchComponent,
FlightCardComponent,

// anwendungsfallübergreifend
DateComponent,
CityPipe,
StatusColorPipe,
StatusFilterPipe
],
providers: [],
bootstrap: [
  AppComponent
]
})
export class AppModule { }

```

Sie können sich sicher vorstellen, dass dieses eine Modul sehr bald recht unübersichtlich wird, wenn die Anwendung wächst. Aus diesem Grund bietet es sich an, die Anwendung in mehrere Module zu gliedern. Die soeben diskutierten drei Aspekte helfen dabei. Der nächste Abschnitt stellt eine Modulstruktur vor, die darauf basiert und sich im Angular-Umfeld bewährt hat.

## Eine Angular-typische Modulstruktur

Angular selbst macht weder Vorgaben zur Strukturierung von Anwendungen noch zum Einsatz von Modulen. Allerdings hat sich die in Abbildung 7-1 gezeigte Struktur bewährt.

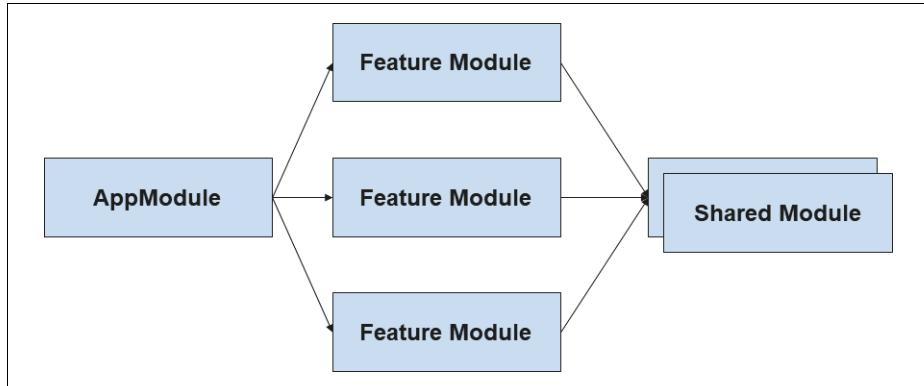


Abbildung 7-1: Typische Modulstruktur

Dieser Struktur zufolge hat die Anwendung ein Root-Module, das die Shell der Anwendung beinhaltet und beim Bootstrapping an Angular übergeben wird. Pro Anwendungsfall weist die Anwendung ein sogenanntes Feature-Module auf. Falls diese Feature-Modules zu groß werden, können sie auch in verschiedene kleinere Module aufgeteilt werden. Als Faustregel bietet es sich an, ein Modul ab etwa 7 +/−2 Einträge zu splitten. Diese Menge entspricht der häufig genannten Anzahl an Dingen, die die meisten Menschen zu einem Zeitpunkt überblicken können.

Außerdem weisen die meisten Anwendungen ein oder mehrere Shared Modules auf. Das sind Module mit allgemeinen Inhalten, die anwendungsfallübergreifend zum Einsatz kommen. Beispiele dafür sind Logging, Authentifizierung, allgemeine Komponenten und Validierungsregeln sowie allgemeine Pipes.

Während sich die Einteilung der geschaffenen Module in diese drei Kategorien aus organisatorischen Gründen bewährt hat, existiert auf technischer Ebene kein Unterschied: Angular kennt lediglich Module, die weitere Module importieren.



Verschiedene Autoren haben im Laufe der Zeit noch weitere Kategorien von Modulen definiert. In älteren Werken findet man zum Beispiel häufig ein sogenanntes Core-Module mit systemweit verwendeten Services. Beispiele dafür sind Services für die Authentifizierung oder fürs Logging.

Da so ein Modul rasch zur »allwissenden Müllhalde« wird und sich mittlerweile die technischen Gründe, die dafür sprechen, vermeiden lassen, raten wir davon ab. Stattdessen empfehlen wir dezidierte Shared Modules, wie ein AuthModule oder ein LoggerModule.

Kapitel 13 erklärt die technischen Gründe, die Core-Modules teilweise rechtfertigen, aber auch Möglichkeiten, diese technischen Gründe zu vermeiden.

## Shared Modules

Lassen Sie uns mit der Einrichtung eines Shared Module beginnen. Dieses lässt sich mit der Angular CLI einrichten:

```
ng generate module shared
```

Die Kurzform für diese Anwendung lautet:

```
ng g m shared
```

Wie gewohnt, lässt sich diese Aufgabe auch mit einem Kontextmenüeintrag des gewünschten Ordners in Visual Studio bewerkstelligen, sofern Sie das Plug-in *Angular Schematics* installiert haben (siehe Abbildung 7-2). Die danach vom Plug-in gestellten Fragen können Sie getrost mit den vorgeschlagenen Werten durch Drücken von *Enter* beantworten.

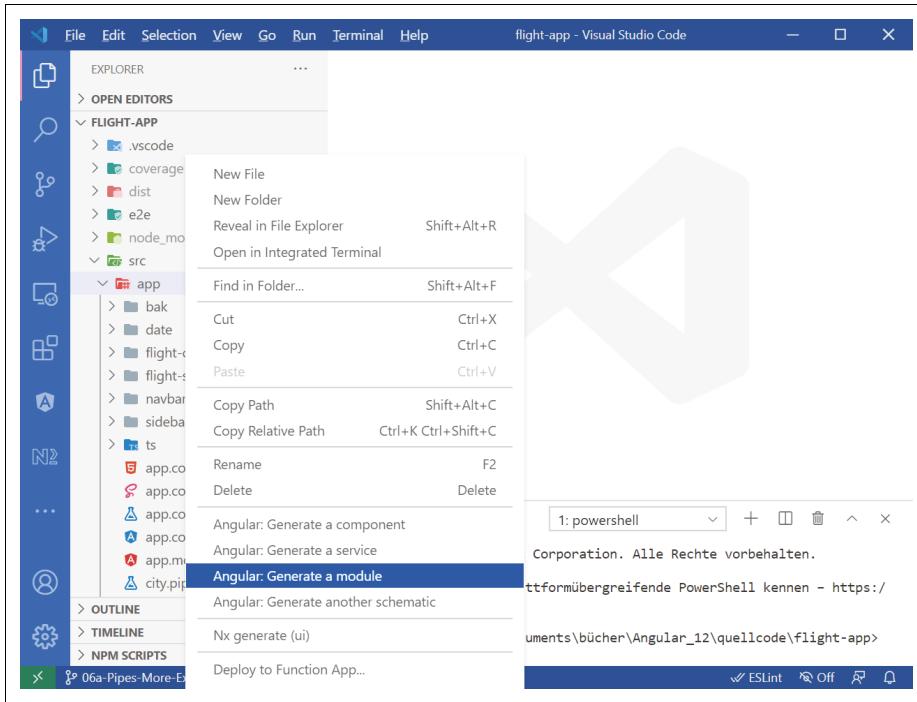


Abbildung 7-2: Modul in Visual Studio Code mit dem Plug-in Angular Schematics generieren

Daraufhin erhalten Sie einen neuen Ordner *shared* mit einer *shared.module.ts*. Das darin zu findende Grundgerüst unseres Moduls gestaltet sich wie in Beispiel 7-2.

#### Beispiel 7-2: Grundgerüst für das SharedModule

```
// src/app/shared/shared.module.ts

import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

@NgModule({
  declarations: [],
  imports: [
    CommonModule
  ]
})
export class SharedModule { }
```



Kleine Anwendungen kommen oft mit einem Shared Module aus, das häufig – so wie hier – den Namen *SharedModule* bekommt. Werden die Anwendung und somit auch dieses Modul größer, bietet es sich an, es auf mehrere Shared Modules aufzusplitten. Diese erhalten dann sprechendere Namen, wie *AuthModule*, *LoggerModule* oder *ValidationModule*.

Das erhaltene Grundgerüst importiert auch schon das `CommonModule`. Es beinhaltet die Building-Blocks, die Angular zur Verfügung stellt. Beispiele dafür sind `ngFor`, `ngIf`, `ngStyle` und `ngClass`, aber auch Built-in-Pipes, wie `date` oder `json`.

Eventuell fragen Sie sich, warum Sie dieses Modul erst jetzt zum ersten Mal sehen. Schließlich haben wir all diese Building-Blocks bereits verwendet. Die Antwort ist, dass das von unserem `AppModule` importierte `BrowserModule` sämtliche Building-Blocks des `CommonModule` veröffentlicht. Allerdings darf das `BrowserModule` nur einmal importiert werden. Deswegen verweisen alle weiteren Module auf das `CommonModule`.

Es hat sich in der Angular-Community eingebürgert, alle Bestandteile eines Moduls in dessen Ordner zu platzieren. Diese Vorgehensweise hilft dabei, die Übersicht zu bewahren. In unserem Fall bedeutet das, dass wir nun die in Abbildung 7-3 gezeigten Dateien in den Ordner `shared` verschieben.

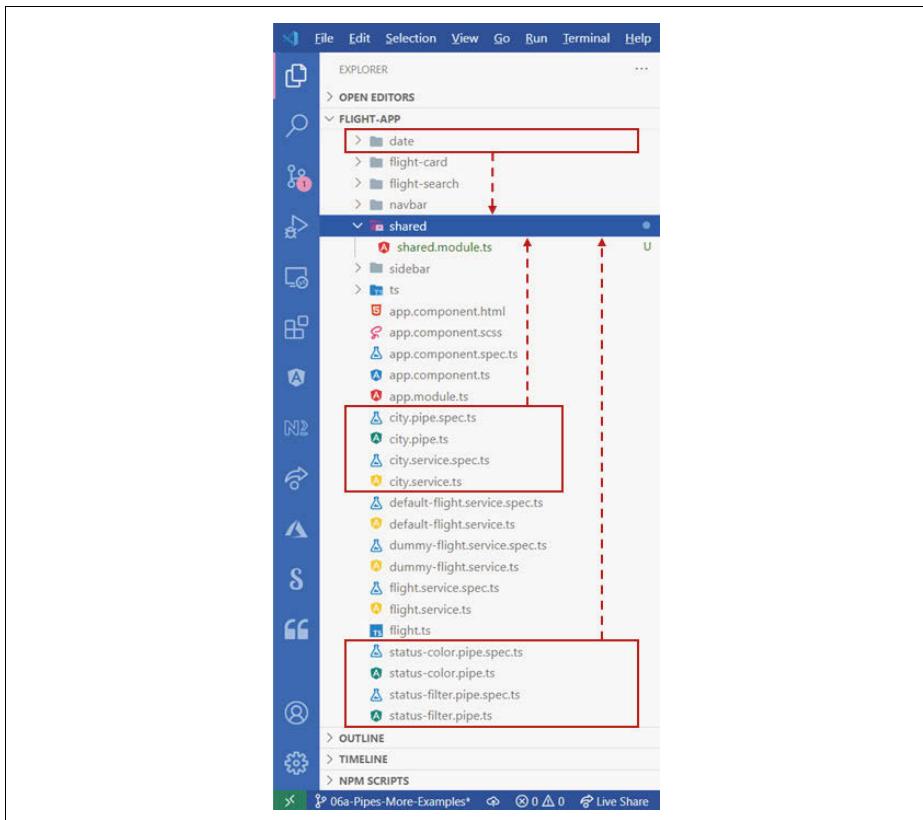


Abbildung 7-3: Building-Blocks nach `shared` verschieben

Falls Sie nicht alle bisher gezeigten Beispiele umgesetzt haben, ist das kein Problem. In diesem Fall verschieben Sie einfach die von Ihnen implementierte Untermenge. Dasselbe gilt für alle weiteren Abschnitte dieses Kapitels.



Verschieben Sie die Dateien direkt in Visual Studio Code. Die Entwicklungsumgebung versucht in diesem Fall, alle nötigen import-Anweisungen zu aktualisieren.

In manchen Fällen schafft Visual Studio Code das leider nicht. In diesem Fall müssen Sie die import-Anweisungen manuell korrigieren. Glücklicherweise weist der TypeScript-Compiler auf die entsprechenden Stellen hin.

Nun können wir die in den Ordner *shared* verschobenen Building-Blocks beim SharedModule registrieren (siehe Beispiel 7-3).

*Beispiel 7-3: SharedModule mit Declarations und Exports*

```
// src/app/shared/shared.module.ts
```

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { DateComponent } from './date/date.component';
import { CityPipe } from './city.pipe';
import { StatusColorPipe } from './status-color.pipe';
import { StatusFilterPipe } from './status-filter.pipe';
import { FormsModule } from '@angular/forms';

@NgModule({
  imports: [
    CommonModule,
    FormsModule
  ],
  declarations: [
    DateComponent,
    CityPipe,
    StatusColorPipe,
    StatusFilterPipe
  ],
  exports: [
    DateComponent,
    CityPipe,
    StatusColorPipe,
    StatusFilterPipe
  ]
})
export class SharedModule { }
```

Bitte beachten Sie, dass wir die generierte Reihenfolge von imports und declarations geändert haben. Aus unserer Sicht ist diese Reihenfolge – zuerst importieren und dann erst deklarieren – logischer, aber das mag Geschmackssache sein. Bei den Importen haben wir das FormsModule ergänzt, da dieses von der DateComponent zur Datenbindung mittels ngModel benötigt wird.

Die declarations verweisen nun auf die Building-Blocks. Somit sind sie nun Teil des SharedModule und lassen sich innerhalb des SharedModule verwenden.

Damit sich die Building-Blocks auch in anderen Modulen nutzen lassen, sind sie zusätzlich unter exports einzutragen. Jedes andere Modul, das nun das SharedModule importiert, bekommt Zugriff auf diese Exporte.



Bitte denken Sie daran, dass sich Services, die auf Modulebene registriert werden, anders verhalten. Sie sind global, sofern es sich nicht um ein lazy Modul handelt, und werden deswegen nicht exportiert. Daher muss das HttpClientModule, das den HttpClient anbietet, auch nur ins AppModule importiert werden.

Weitere Details dazu finden Sie in Kapitel 5.

## Feature-Modules

Lassen Sie uns nun ein Feature-Module für die Flugbuchung einrichten:

```
ng generate module flight-booking
```

Alternativ dazu können Sie auch hier wieder auf das Plug-in *Angular Schematics* in Visual Studio Code zurückgreifen.

Ähnlich wie vorhin schafft dieses Vorgehen einen Ordner *flight-booking* mit einer Datei *flight-booking.module.ts*. Alle Building-Blocks, die wir mit *flight-booking* assoziieren, sind nun in den neuen Ordner zu verschieben (siehe Abbildung 7-4).

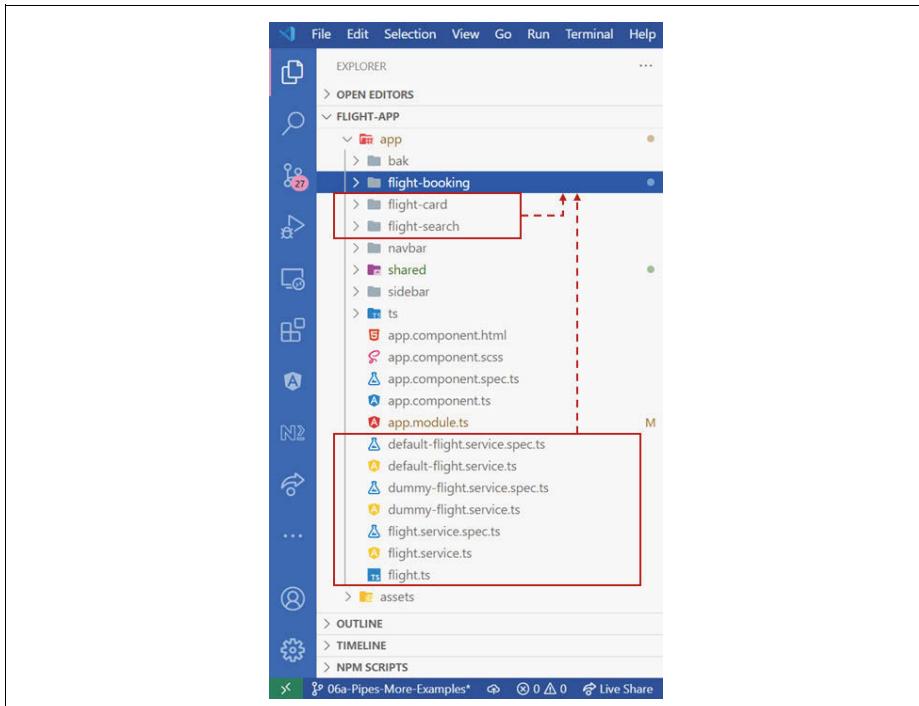


Abbildung 7-4: Building-Blocks nach *flight-booking* verschieben (Details können abweichen)

Sollten Sie nun Kompilierungs- oder Laufzeitfehler bekommen, können Sie diese vorerst ignorieren. Die CLI kommt durch das Verschieben von Dateien durcheinander, und gegebenenfalls müssen Sie auch noch den einen oder anderen Import korrigieren. Sie können auch `ng serve` schließen.



Natürlich kann man hier argumentieren, dass das Flight-Interface und die Services anwendungsfällig übergreifend sind und deswegen in ein eigenes Modul gehören. Das wäre dann zum Beispiel ein `FlightApiModule`, das laut unserem Gedankenmodell den Shared Modules zuzuordnen ist.

Generell versuchen wir, Building-Blocks so lokal wie möglich und so global wie nötig zu positionieren: Da wir in unserer Demoanwendung den Flight und die Services nur in `flight-booking` benötigen, legen wir sie dort ab. Ansonsten würden wir das erwähnte `FlightApi` Module einrichten.

Das entspricht auch einer allgemeinen Vorgehensweise, die sich mittlerweile in der Angular-Community eingebürgert hat: Die Anwendung wird so weit wie sinnvoll möglich nach Features gruppiert.

Nach dem Verschieben der Building-Blocks sind diese in das generierte `FlightBooking` Module einzutragen (siehe Beispiel 7-4).

*Beispiel 7-4: Feature-Module mit Importen, Deklarationen und Exporten*

```
// src/app/flight-booking/flight-booking.module.ts

import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';
import { SharedModule } from '../shared/shared.module';
import { FlightSearchComponent } from './flight-search/flight-search.component';
import { FlightCardComponent } from './flight-card/flight-card.component';

@NgModule({
  imports: [
    CommonModule,
    FormsModule,
    SharedModule
  ],
  declarations: [
    FlightSearchComponent,
    FlightCardComponent
  ],
  exports: [
    FlightSearchComponent
  ]
})
export class FlightBookingModule { }
```

Auch hier haben wir wieder die Reihenfolge der generierten Abschnitte imports und declarations getauscht. Neben dem CommonModule importieren wir hier das FormsModule und das SharedModule. Von Ersterem benötigen wir ngModel für das Two-Way-Binding, von Letzterem brauchen wir die exportieren Pipes sowie die DateComponent.

Zu den declarations zählen die FlightSearchComponent und die davon aufgerufene FlightCardComponent. Da die FlightCardComponent nur modulintern verwendet wird, müssen wir sie nicht exportieren. Somit beschränken sich die exports auf die FlightSearchComponent, die wir in der AppComponent aufrufen.

## Root-Modules

Durch das Auslagern der einzelnen Building-Blocks in separate Module können wir nun unser AppModule deutlich schlanker gestalten (siehe Beispiel 7-5).

*Beispiel 7-5: Das nun schlanke AppModule*

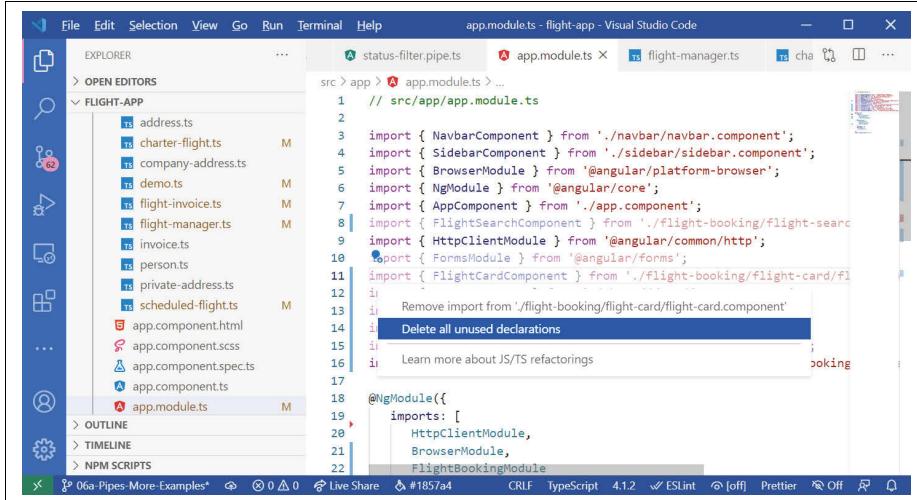
```
// src/app/app.module.ts

import { NavbarComponent } from './navbar/navbar.component';
import { SidebarComponent } from './sidebar/sidebar.component';
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
import { HttpClientModule } from '@angular/common/http';
import { FlightBookingModule } from './flight-booking/flight-booking.module';

@NgModule({
  imports: [
    HttpClientModule,
    BrowserModule,
    FlightBookingModule
  ],
  declarations: [
    AppComponent,
    SidebarComponent,
    NavbarComponent
  ],
  providers: [],
  bootstrap: [
    AppComponent
  ]
})
export class AppModule { }
```



Visual Studio Code entfernt auf Wunsch sämtliche nicht benötigten Importe:



The screenshot shows the Visual Studio Code interface with the 'app.module.ts - flight-app - Visual Studio Code' tab active. In the Explorer sidebar, there's a tree view of files under 'FLIGHT-APP'. The 'src/app/app.module.ts' file is open in the editor. A cursor is placed on a line that imports 'FlightBookingModule'. A context menu is open at this position, with the option 'Delete all unused declarations' highlighted in blue. Other options like 'Learn more about JS/TS refactorings' and 'Remove import from ...' are also visible. The status bar at the bottom shows 'Live Share' and other development settings.

Platzieren Sie dazu den Cursor in einer Zeile mit einem nicht verwendeten Import. Klicken Sie danach auf das erscheinende blaue Icon oder drücken Sie *Strg+.* (Punkt). Nun bietet Ihnen Visual Studio Code die Option *Delete all unused declarations* an, die sämtliche nicht benötigten Importe verschwinden lässt.

Wenn Sie nun Ihre Anwendung neu starten (`ng serve -o`), sollte die Anwendung wie gewohnt funktionieren. Es kann sein, dass Sie an dieser Stelle Kompilierungs- oder Laufzeitfehler erhalten. Diese hängen sehr wahrscheinlich mit fehlerhaften Importen zusammen, die sich durch das Verschieben der Building-Blocks ergeben haben. Korrigieren Sie die Stellen, auf die Sie der Compiler hinweist, und starten Sie gegebenenfalls `ng serve -o` erneut.

Auch wenn sich an der Funktionalität unserer Anwendung in diesem Kapitel nichts geändert hat, war die beschriebene Restrukturierung wichtig: Sie erlaubt der Anwendung, nach und nach zu wachsen, und hilft uns, den Überblick zu wahren.

## Module reexportieren

Manche Module werden in so gut wie allen anderen Modulen benötigt. Ein Beispiel dafür ist das `CommonModule`. Um diese Module nicht manuell überall importieren zu müssen, bietet sich der Einsatz eines Sammelmoduls an. Mit diesem nicht offiziellen Begriff bezeichnen wir Module, die ganze andere Module exportieren. Ein Beispiel ist das aktualisierte `SharedModule` in Beispiel 7-6.

*Beispiel 7-6: Das SharedModule exportiert nun andere Module.*

```
// src/app/shared/shared.module.ts

import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { DateComponent } from './date/date.component';
import { CityPipe } from './city.pipe';
import { StatusColorPipe } from './status-color.pipe';
import { StatusFilterPipe } from './status-filter.pipe';

@NgModule({
  imports: [
    CommonModule,
  ],
  declarations: [
    DateComponent,
    CityPipe,
    StatusColorPipe,
    StatusFilterPipe
  ],
  exports: [
    DateComponent,
    CityPipe,
    StatusColorPipe,
    StatusFilterPipe,
    // Hinzufügen:
    CommonModule
  ]
})
export class SharedModule { }
```

Das SharedModule exportiert nun das CommonModule. Das bedeutet, dass jedes Modul, das das SharedModule importiert, auch Zugriff auf das CommonModule erhält. Somit lässt sich unser FlightBookingModule wie in Beispiel 7-7 gezeigt vereinfachen.

*Beispiel 7-7: Das FlightBookingModule erhält nun das CommonModule und das FormsModule über das SharedModule.*

```
// src/app/flight-booking/flight-booking.module.ts

[...]

@NgModule({
  imports: [
    // Diesen Import entfernen:
    // CommonModule,
    FormsModule,
    SharedModule
  ],
  declarations: [
    FlightSearchComponent,
    FlightCardComponent
  ],
})
```

```
exports: [
  FlightSearchComponent
]
})
export class FlightBookingModule { }
```

## Zusammenfassung

Module fassen zusammengehörige Building-Blocks wie Komponenten oder Pipes zusammen und helfen somit beim Strukturieren einer Angular-Anwendung.

Typischerweise enthält eine Anwendung neben dem AppModule, das als Hauptmodul (Root-Module) sämtliche Komponenten für den Programmstart und die Shell bereitstellt, pro Feature ein eigenes sogenanntes Feature-Module. Funktionalitäten, die Sie in mehreren Feature-Modules benötigen, können Sie in sogenannte Shared Modules auslagern.

Jedes Modul bekommt einen eigenen Ordner mit einer Datei, die das Modul beschreibt – z.B. *flight-booking.module.ts*. Auch alle Building-Blocks dieses Moduls werden in diesem Ordner oder in entsprechenden Unterordnern verstaut. Das hilft dabei, die Übersicht zu wahren.

# KAPITEL 8

# Routing

Eine *Single Page Application* (SPA) besteht, wie der Name schon ausdrückt, aus nur einer Seite. Um verschiedene Anwendungsfälle anbieten zu können, müssen wir verschiedene Seiten simulieren. Das erfolgt durch das Ein- und Ausblenden von Komponenten. Der Angular-Router hilft bei dieser Aufgabe.

Dieses Kapitel ergänzt unser Beispiel, sodass es unter Nutzung des Angular-Routers mehrere Ansichten präsentiert. Diese sogenannten Routen lassen sich über einzelne Menüeinträge einblenden.

## Überblick

Wenn eine SPA mehrere Seiten simulieren soll, reicht es nicht, einfach nur Komponenten ein- und auszublenden. Damit der Back-Button funktioniert, muss sich der durchgeführte Zustandswechsel in der URL widerspiegeln. Dasselbe gilt für Bookmarks oder Links, die auf eine bestimmte Ansicht der SPA verweisen. Glücklicherweise automatisiert der Router auch diese Aufgabe, die man ebenfalls als *Deep Linking* bezeichnet: Er spendiert jeder Route eine eigene URL.

Der Router, der im Lieferumfang von Angular enthalten ist, sieht vor, dass die SPA neben konkreten Bereichen, wie Menüs oder Fußzeilen, auch einen Platzhalter aufweist (siehe Abbildung 8-1).

Um festzulegen, welche Komponente der Router in diesem Platzhalter positionieren soll, hängt der Aufrufer einen zusätzlichen Pfad an die URL an. Dieser Pfad verweist auf einen Konfigurationseintrag, der unter anderem die Komponente bekannt gibt. Man sagt auch, dass der Router die adressierte Komponente aktiviert; Abbildung 8-2 veranschaulicht dies. Hier wurde an die URL der SPA der Pfad `/flug-suchen` angehängt. Das veranlasst den Router, die damit assoziierte FlightSearchComponent zu aktivieren.

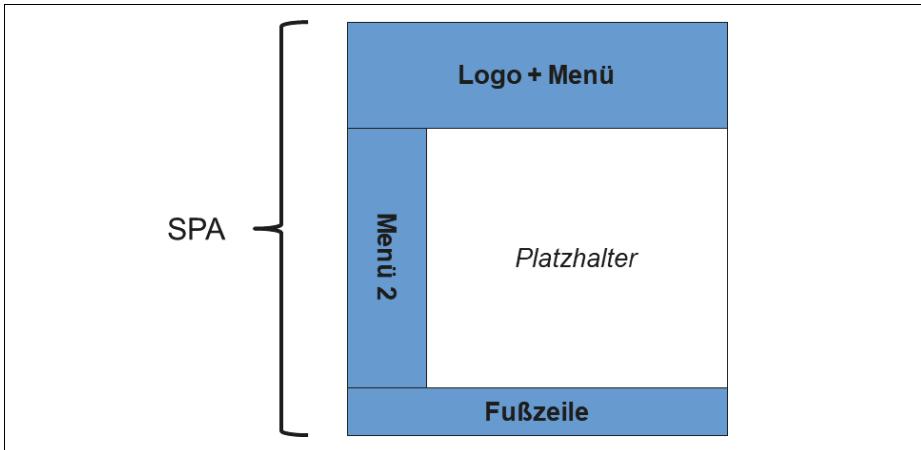


Abbildung 8-1: SPA mit Platzhalter für das Routing

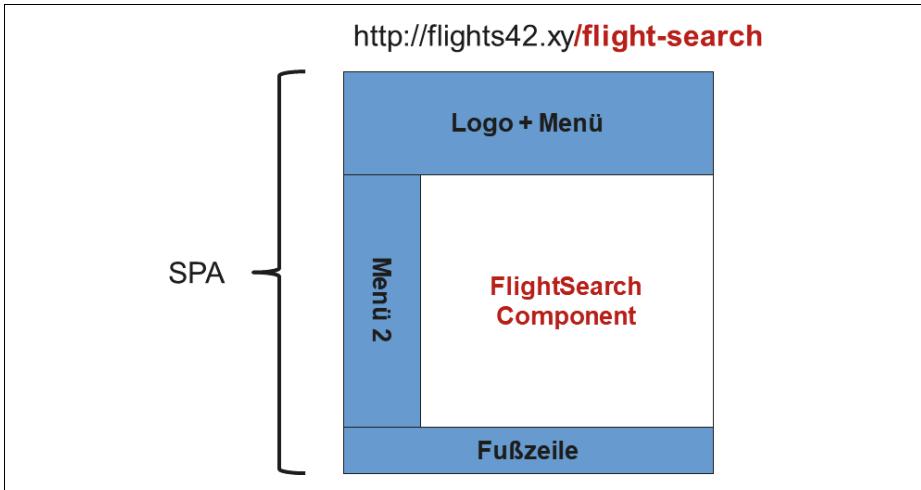


Abbildung 8-2: Aktivieren von Komponenten mit dem Angular-Router

Um die Übersicht zu wahren, finden wir in einer Angular-Anwendung typischerweise mehrere Routing-Konfigurationen: eine für das AppModule und eine weitere für die einzelnen Feature-Modules.

Um eine Konfiguration für eine Route zu finden, durchsucht der Router zunächst die Konfigurationen der Feature-Module in jener Reihenfolge, in der sie *importiert* wurden. Am Ende durchsucht er die Konfiguration des AppModule. Der erste Eintrag, der zur aktuellen URL passt, wird herangezogen, und dessen Komponente wird im Platzhalter angezeigt.

Deswegen ist auch die Reihenfolge der Routen und Routenkonfigurationen ausschlaggebend. Idealerweise nutzt jede Route eine eindeutige URL, sodass sich Konflikte von Anfang an vermeiden lassen.

# Erste Schritte mit dem Router

Um die Funktionsweise des Routers zu veranschaulichen, werden wir endlich die Menübefehle auf der linken Seite an unsere Bedürfnisse anpassen und mit Leben füllen (siehe Abbildung 8-3).

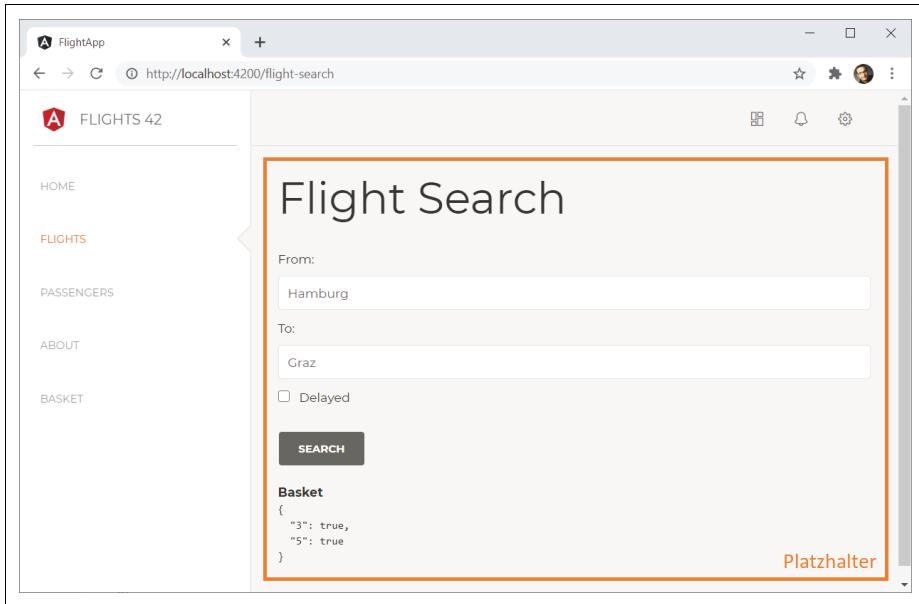


Abbildung 8-3: Der Router hat die FlightSearchComponent in den Platzhalter geladen.

Der Platzhalter befindet sich in dieser Anwendung rechts vom Seitenmenü. Er soll abhängig vom Anwendungszustand eine der folgenden Komponenten anzeigen:

- HomeComponent: Zeigt eine Begrüßung an.
- FlightSearchComponent: Unsere Komponente zum Suchen nach Flügen.
- PassengerSearchComponent: Komponente zum Suchen nach Passagieren. Vor erst handelt es sich dabei lediglich um eine Dummy-Komponente, die als weiteres Routing-Ziel fungiert.
- AboutComponent: Zeigt allgemeine Informationen zur Anwendung.
- NotFoundComponent: Wird angezeigt, wenn die gewünschte Route nicht gefunden wurde.

Diese – bis auf die FlightSearchComponent – neuen Komponenten können Sie wie gewohnt mit der Angular CLI erzeugen:

```
ng generate component home
ng generate component flight-booking/passenger-search
ng generate component about
ng generate component not-found
```

Bitte beachten Sie, dass die zweite Anweisung die PassengerSearchComponent im Ordner *flight-booking* erzeugt. Darin befindet sich unser FlightBookingModule, bei dem die CLI die Komponente auch registriert. Alle anderen Komponenten erzeugt die CLI im Ordner *app* und registriert sie bei der sich dort befindlichen AppComponent.

Anstatt die CLI auf der Konsole zu nutzen, können Sie auch auf das bereits besprochene Plug-in *Angular Schematics* in Visual Studio Code zurückgreifen.



Prüfen Sie zur Sicherheit, ob die Angular CLI die generierten Komponenten erfolgreich beim AppModule bzw. FlightBookingModule registriert hat.

Da unsere Benutzer eine ordentliche Begrüßung verdienen, haben wir das Template der HomeComponent entsprechend abgeändert:

```
<!-- src/app/home/home.component.html -->
<h1>Welcome!</h1>
```

## Routing-Konfiguration für das AppModule einrichten

Damit der Router weiß, wann welche Komponente zu aktivieren ist, stellen wir ihm im Ordner *src/app* die Datei *app.routes.ts* mit einer Routing-Konfiguration für die Komponenten im AppModule bereit. Für die Routen, die zu Komponenten unseres FlightBookingModule führen, erzeugen wir später eine weitere Konfiguration.

Bei einer Routing-Konfiguration handelt es sich um eine herkömmliche TypeScript-Datei, die sich direkt in Visual Studio Code erzeugen lässt (Rechtsklick auf den Ordner *app/New File*). Darin befindet sich eine Array-Konstante mit Objekten vom Typ Route, die in erster Linie Pfade auf Komponenten abbilden (siehe Beispiel 8-1).

*Beispiel 8-1: Routing-Konfiguration für das AppModule*

```
// src/app/app.routes.ts

import { Routes } from '@angular/router';
import { AboutComponent } from './about/about.component';
import { HomeComponent } from './home/home.component';
import { NotFoundComponent } from './not-found/not-found.component';

export const APP_ROUTES: Routes = [
  {
    // Standardroute: Umleitung auf '/home'
    path: '',
    redirectTo: 'home',
    pathMatch: 'full'
  },
  {
    path: 'home',
    component: HomeComponent
  },
  {
    path: 'about',
    component: AboutComponent
  },
  {
    path: '**',
    component: NotFoundComponent
  }
];
```

```

    {
      path: 'about',
      component: AboutComponent
    },
    {
      path: '**',
      component: NotFoundComponent
    }
];

```

Etwas Aufmerksamkeit verdient hier die erste Route: Diese weist keinen Pfad auf und fungiert deswegen als Standardroute. Angular aktiviert sie, wenn der Aufrufer keinen Pfad an die URL der SPA anhängt. Ein Beispiel dafür ist `http://localhost:4200`. Mit `redirectTo` leitet die Standardroute auf die darunter definierte home-Route weiter.

Eine kleine Herausforderung gibt es jedoch bei solchen Routen: Standardmäßig prüft Angular nur, ob der Pfad in der Konfiguration (z.B. `path: myRoute`) ein Präfix des Pfads in der URL ist (z.B. `http://localhost:4200/myRoute/something-else`). Dummerweise sieht JavaScript einen Leerstring als Präfix aller anderen Strings an. Somit würde der Router die Standardroute mit leerem Pfad immer heranziehen.

Die Lösung für dieses Problem ist die Eigenschaft `pathMatch: full`. In diesem Fall vergleicht Angular den gesamten Pfad aus der Konfiguration mit dem gesamten Pfad in der URL.

Die Einstellung `path: '**'` im letzten Eintrag bewirkt, dass sämtliche weiteren Pfade zur `NotFoundComponent` führen. Damit schaffen wir ein letztes Auffangnetz.

Damit Angular diese Konfiguration aufgreift, ist sie gemeinsam mit dem Router Module ins AppModule zu importieren (siehe Beispiel 8-2).

#### *Beispiel 8-2: RouterModule ins AppModule importieren*

```

// src/app/app.module.ts

[...]
// Diese beiden Importe einfügen:
import { RouterModule } from '@angular/router';
import { APP_ROUTES } from './app.routes';

@NgModule({
  imports: [
    // Diese Zeilen hinzufügen:
    RouterModule.forRoot(APP_ROUTES),
    [...]
  ],
  declarations: [
    [...]
  ],
  providers: [],
  bootstrap: [

```

```

        AppComponent
    ]
})
export class AppModule { }

```

Bitte beachten Sie, dass wir hier die Routing-Konfiguration an `RouterModule.forRoot` übergeben. Da diese Methode systemweite Services einrichtet, darf sie nur im AppModule aufgerufen werden. Um weitere Routenkonfigurationen für die einzelnen Feature-Modules zu registrieren, kommt stattdessen die Methode `RouterModule.forChild` zum Einsatz. Der nächste Abschnitt veranschaulicht das.



Die Methode `forRoot` akzeptiert neben der Routenkonfiguration auch noch einen zweiten Parameter, der ein Konfigurationsobjekt entgegennimmt. Mit diesem Objekt, das zahlreiche Eigenschaften besitzt, lässt sich das Verhalten des Routers anpassen. Eine dieser Eigenschaften kann Ihnen beim Lösen von Problemen helfen:

```
RouterModule.forRoot(APP_ROUTES, { enableTracing: true }),
```

Setzen Sie `enableTracing` auf `true`, informiert Sie der Router über seine Tätigkeiten. Dazu gibt er entsprechende Meldungen in der JavaScript-Konsole Ihres Browsers aus.

## Routing-Konfiguration für Feature-Modules einrichten

Nachdem wir Routen für die Komponenten im AppModule eingerichtet haben, müssen wir diese Aufgabe für das FlightBookingModule wiederholen. Für die Konfiguration richten wir im Ordner `src/app/flight-booking` eine Datei `flight-booking.routes.ts` ein (siehe Beispiel 8-3).

### *Beispiel 8-3: Routenkonfiguration für Feature-Modules*

```
// src/app/flight-booking/flight-booking.routes.ts

import { Routes } from '@angular/router';
import { FlightSearchComponent } from './flight-search/flight-search.component';
import { PassengerSearchComponent } from './passenger-search/passenger-
search.component';

export const FLIGHT_BOOKING_ROUTES: Routes = [
  {
    path: 'flight-search',
    component: FlightSearchComponent
  },
  {
    path: 'passenger-search',
    component: PassengerSearchComponent
  }
];
```

Diese Konfiguration ist wie jene für das AppModule aufgebaut und bildet Pfade auf Komponenten ab. Diese Konfiguration ist nun beim FlightBookingModule zu registrieren (siehe Beispiel 8-4).

*Beispiel 8-4: RouterModule inklusive Konfiguration in FlightBookingModule importieren*

```
// src/app/flight-booking/flight-booking.module.ts

[...]

// Diese beiden Importe hinzufügen:
import { RouterModule } from '@angular/router';
import { FLIGHT_BOOKING_ROUTES } from './flight-booking.routes';

@NgModule({
  imports: [
    // Diesen Eintrag hinzufügen:
    RouterModule.forChild(FLIGHT_BOOKING_ROUTES),
    SharedModule
  ],
  declarations: [
    FlightSearchComponent,
    FlightCardComponent,
    PassengerSearchComponent
  ],
  exports: [
    FlightSearchComponent
  ]
})
export class FlightBookingModule { }
```

Das Beispiel übergibt die Routenkonfiguration an RouterModule.forChild. Diese Methode ist bei allen Feature-Modules zu nutzen. Lediglich beim AppModule kommt stattdessen RouterModule.forRoot zum Einsatz.

## Platzhalter in AppComponent hinterlegen

Anstatt auf eine konkrete Komponente zu verweisen, nutzt die AppComponent nun einen Platzhalter. Diesen repräsentiert der Router durch ein router-outlet-Element (siehe Beispiel 8-5).

*Beispiel 8-5: Template der AppComponent mit router-outlet-Element*

```
<!-- src/app/app.component.html -->

<div class="wrapper">

  <div class="sidebar" data-color="white" data-active-color="danger">
    <app-sidebar-cmp></app-sidebar-cmp>
  </div>

  <div class="main-panel">
    <app-navbar-cmp></app-navbar-cmp>

    <div class="content">

      <!-- Diese Zeile entfernen: -->
      <!-- <app-flight-search></app-flight-search> -->
```

```

<!-- Diese Ziele hinzufügen: -->
<router-outlet></router-outlet>

</div>
</div>

</div>

```

## Hyperlinks zum Aktivieren von Routen einrichten

Nun benötigen wir nur noch Hyperlinks, die die einzelnen Routen im Platzhalter aktivieren. Dazu passen wir die generierte SidebarComponent an, wie in Beispiel 8-6 gezeigt.

*Beispiel 8-6: Verweise auf Routen im Template der SidebarComponent*

```

<!-- src/app/sidebar/sidebar.component.html -->

[...]

<!-- Diese Einträge um routerLink -->
<!-- und routerLinkActive erweitern: -->
<li routerLinkActive="active">
    <a routerLink="home">
        <p>Home</p>
    </a>
</li>

<li routerLinkActive="active">
    <a routerLink="flight-search" >
        <p>Flights</p>
    </a>
</li>

<li routerLinkActive="active">
    <a routerLink="passenger-search">
        <p>Passengers</p>
    </a>
</li>

<!-- Diesen Eintrag ergänzen: -->
<li routerLinkActive="active">
    <a routerLink="about">
        <p>About</p>
    </a>
</li>

[...]

```

Die aus dem RouterModule stammende Direktive `routerLink` verweist auf die Pfade der konfigurierten Routen. Die Direktive `routerLinkActive` verweist hingegen auf eine Klasse, mit deren Stylings der aktive Menüpunkt hervorgeben wird. Standardmäßig weist sie die Klasse dem Element zu, wenn das Element einen aktiven router Link aufweist oder dies auf ein Child-Element zutrifft.



Sollten Sie in AngularJS-1.x-Anwendungen mit dem UI-Router gearbeitet haben, fragen Sie sich eventuell, ob `routerLink` auch auf einen internen Routennamen verweisen kann, um die Angabe der Pfade nicht wiederholen zu müssen. Leider sieht der Angular-Router diese Indirektion nicht vor.

Wenn Sie nun Ihre Anwendung starten, sollten die Menüeinträge auf der linken Seite auf die einzelnen Routen verweisen (siehe Abbildung 8-4).

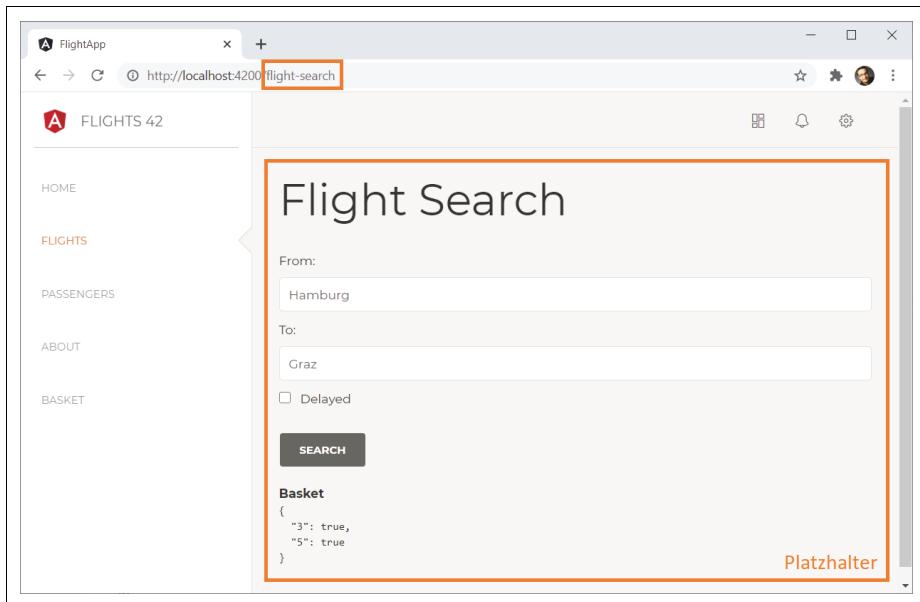


Abbildung 8-4: Routing in der Demoanwendung

Die aktuelle Route wird nun auch durch die URL in der Adresszeile widergespiegelt.



Statt mit Hyperlinks können Sie einen Routenwechsel auch programmatisch anstoßen. Injizieren Sie dazu den Router:

```
[...]
import { Router } from '@angular/router';

@Component({ [... ] })
export class AppComponent {
    constructor(private router: Router) { }

    goHome(): void {
        this.router.navigate(['/home']);
    }
}
```

Die Methode `navigate` nimmt den Pfad der gewünschten Route als Array entgegen. Jeder Array-Eintrag entspricht einem URL-Segment.

Diese werden URL-codiert und zusammengekettet. Der sich so ergebende Pfad wird zur Identifizierung der Zielroute verwendet. Der fiktive Aufruf

```
this.router.navigate(['/flights', 'Berlin', id]);
```

führt somit zur Aktivierung der Route `/flights/Berlin/17`, wenn man davon ausgeht, dass die Variable `id` den Wert `17` hat.

## Parametrisierte Routen

Beim Routenwechsel gilt es häufig, Informationen an die adressierten Routen zu übergeben. Beispielsweise muss die Anwendung einer Route zum Editieren eines Flugs dessen ID mitteilen. Hierzu kommen Routing-Parametern zum Einsatz.

### Arten von Routing-Parametern

Für die Angabe von Routing-Parametern innerhalb der URL unterstützt Angular verschiedene Schreibweisen (siehe Tabelle 8-1).

Tabelle 8-1: Schreibweisen für Routing-Parameter

Position	Beispiel
URL-Segment	<code>flight-booking/flight-edit/17</code>
Matrixparameter	<code>flight-booking/flight-edit/17;showDetails=true;airline=LH</code>
Query-String	<code>flight-booking/flight-edit/17?showDetails=true&amp;airline=LH</code>

Parameter, die in ein URL-Segment platziert werden, erkennt Angular anhand ihrer Position. Bei Matrixparametern und dem Query-String ist die Position der Parameter nicht ausschlaggebend, zumal hier ohnehin der Parametername bekannt gegeben wird.

Bei solchen benannten Parametern verwendet der Router standardmäßig Matrixparameter. Im Gegensatz zum besser bekannten Query-String bezieht sich ein Matrixparameter immer auf das aktuelle URL-Segment. Somit kann der Router solche Parameter einem Segment und somit auch einer Komponente zuordnen. Das folgende Beispiel macht auf diese Weise klar, dass sich `orderBy` auf das Segment `passagiere` bezieht und die beiden anderen Parameter auf `flug-edit/17`:

```
flug-buchen/flug-edit/17;showDetails=true;airline=LH/passagiere;orderBy=Name
```

### Parameter in Komponenten auslesen

Um den Einsatz parametrisierter Routen zu demonstrieren, führen wir in diesem Abschnitt eine `FlightEditComponent` ein, die die Parameter `id` und `showDetails` entgegennimmt (siehe Abbildung 8-5).

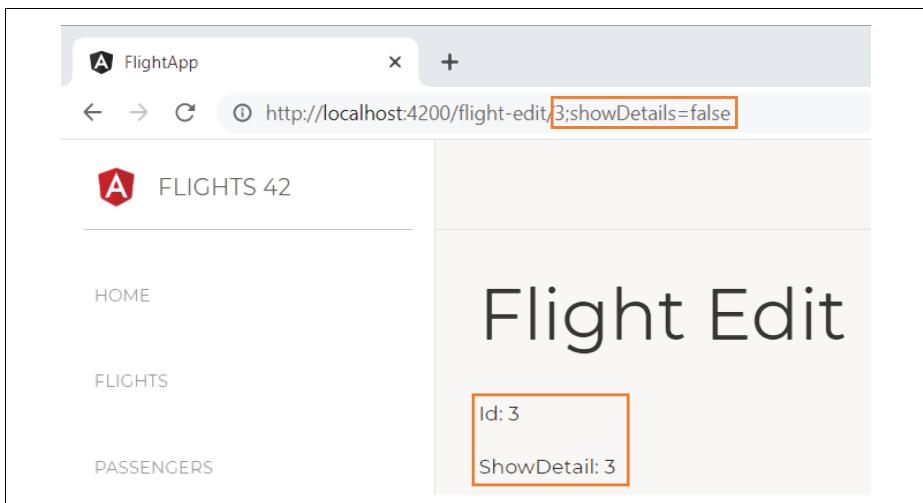


Abbildung 8-5: Die FlightEditComponent nimmt Routing-Parameter entgegen.

Während sich der Parameter `id` in einem eigenen URL-Segment befindet, handelt es sich bei `showDetails` um einen Matrixparameter.

Der Vollständigkeit halber platzieren wir hier den CLI-Aufruf zum Generieren der neuen `FlightEditComponent`, die im Ordner `flight-booking` und somit für das `Flight BookingModule` eingerichtet wird:

```
ng generate component flight-booking/flight-edit
```

Um die Routing-Parameter auslesen zu können, fordert die `FlightEditComponent` den Service `ActivatedRoute` an (siehe Beispiel 8-7).

Beispiel 8-7: Die `ActivatedRoute` stellt die Routing-Parameter zur Verfügung.

```
// src/app/flight-booking/flight-edit/flight-edit.component.ts
```

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

@Component({
  selector: 'app-flight-edit',
  templateUrl: './flight-edit.component.html',
  styleUrls: ['./flight-edit.component.scss']
})
export class FlightEditComponent implements OnInit {

  id = 0;
  showDetails = false;

  constructor(private route: ActivatedRoute) { }

  ngOnInit(): void {
    this.route.params.subscribe(p => {
      this.id = p.id;
    })
  }
}
```

```

        this.showDetails = p.showDetails;
    });
}

```

Die `ActivatedRoute` bietet neben anderen Eigenschaften, die die gerade aktivierte Route beschreiben, ein Observable `params` an. Dieses veröffentlicht sämtliche Routing-Parameter über ein Objekt, das als Dictionary genutzt wird.

Das gezeigte Beispiel liest die beiden Parameter aus und platziert sie in gleichnamigen Eigenschaften. Das Template der Komponente präsentiert diese Eigenschaften (siehe Beispiel 8-8).

*Beispiel 8-8: Präsentation der Routing-Parameter im Template der FlightEditComponent*

```

<h1>Flight Edit</h1>

<p>
  Id: {{id}}
</p>
<p>
  ShowDetail: {{id}}
</p>

```

An dieser Stelle wollen wir uns mit der bloßen Ausgabe der Parameter zufriedengeben. Kapitel 10 wird diese Komponente jedoch um ein Formular zum Verwalten von Flügen erweitern.

## Parametrisierte Routen konfigurieren

Die Konfiguration parametrisierbarer Routen unterscheidet sich nicht wesentlich von jener der bisher betrachteten Routen. Sie müssen lediglich Platzhalter für Parameter, denen eigene URL-Segmente spendiert werden, in den Pfaden vorsehen. Diese Platzhalter sind mit einem Doppelpunkt einzuleiten. Beispiel 8-9 definiert zum Beispiel mit `:id` einen Platzhalter für den Parameter `id`.

*Beispiel 8-9: Konfiguration der parametrisierten Route flight-edit*

```

// src/app/flight-booking/flight-booking.routes.ts

[...]

// Diesen Import hinzufügen:
import { FlightEditComponent } from './flight-edit/flight-edit.component';

export const FLIGHT_BOOKING_ROUTES: Routes = [
  {
    path: 'flight-search',
    component: FlightSearchComponent
  },
  {

```

```

        path: 'passenger-search',
        component: PassengerSearchComponent
    },
    // Diesen Eintrag hinzufügen:
    {
        path: 'flight-edit/:id',
        component: FlightEditComponent
    }
];

```

Benannte Parameter erkennt Angular zur Laufzeit ohne unser Zutun. Deswegen sind sie auch nicht in der Konfiguration zu hinterlegen.

## Auf parametisierte Routen verweisen

Um auf parametisierte Routen zu verweisen, nimmt `routerLink` die einzelnen URL-Segmente, aber auch Matrixparameter als Array entgegen. Beispiel 8-10) erweitert zum Beispiel das Template der `FlightCardComponent` um `routerLink`, der zur zuvor eingeführten `FlightEditComponent` führt.

*Beispiel 8-10: Die Direktive `routerLink` verweist auf die parametisierte Route `flight-edit`.*

```
<!-- src/app/flight-booking/flight-card/flight-card.component.html -->

[...]
<p>
    <button class="btn btn-default" *ngIf="!selected" (click)="select()">
        Select
    </button>

    <button class="btn btn-default" *ngIf="selected" (click)="deselect()">
        Remove
    </button>

    <!-- Diesen Link einfügen: -->
    <a class="btn btn-default"
        [routerLink]="['/..flight-edit', item?.id, showDetails:false]">
        Edit
    </a>
</p>
[...]
```

Die einzelnen Array-Einträge repräsentieren URL-Segmente. Die Direktive `routerLink` führt eine URL-Codierung durch und kettet sie anschließend zu einer URL zusammen. Die Eigenschaften von Objekten werden dabei zu Matrixparametern. Aus dem im betrachteten Beispiel verwendeten Array entsteht somit der folgende Pfad, wenn wir davon ausgehen, dass `item.id` den Wert 3 aufweist:

```
../flight-edit/3;showDetails=true
```

Das Präfix `..` ist notwendig, da wir vorerst davon ausgehen, dass die `FlightCardComponent` von der `FlightSearchComponent` aufgerufen wird. Und ihre Route ist in der Routenkonfiguration einer Schwester von `flight-edit` zu finden (siehe Beispiel 8-9).



Es wäre schöner, wenn die `FlightCardComponent` keine Annahme darüber treffen müsste, wo sich auf welcher Ebene der Routenkonfiguration ihre Parent-Komponente befindet. Das könnte man erreichen, indem man das Präfix für den Pfad, der zur `flight-edit`-Route führt, über ein Property-Binding entgegennimmt.

Man könnte aber auch den gesamten Button beim Aufruf der Komponente übergeben. Hierzu bietet Angular ein Konzept namens Content Projection, das wir in Kapitel 18 näher beschreiben.

Wenn Sie die Lösung nun starten (`ng serve -o`), gelangen Sie über die Schaltfläche `Edit` in den Flugkarten auf die Route `flight-edit` (siehe Abbildung 8-6).

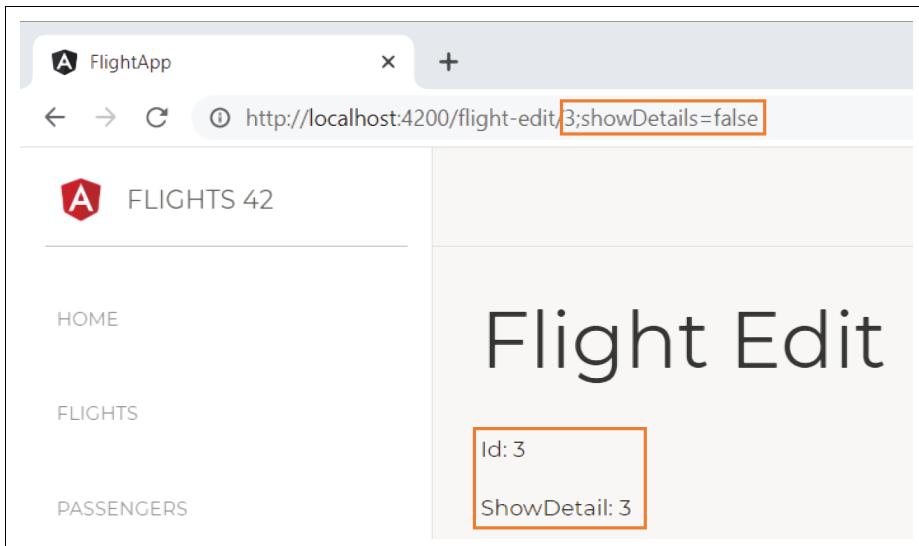


Abbildung 8-6: Die `FlightEditComponent` zeigt die übergebenen Routenparameter an.

## Hierarchisches Routing mit Child-Routes

Komponenten, die der Router aktiviert, können einen weiteren Platzhalter aufweisen. Auf diese Weise lassen sich verschachtelte oder hierarchisch organisierte Views gestalten. Die Routen, die für so einen Platzhalter definiert werden, nennt man auch Child-Routes. In diesem Abschnitt zeigen wir an einem Beispiel die Nutzung dieses Konzepts.

### Überblick über Child-Routes

Ein Beispiel für Child-Routes stellt die `FlightBookingComponent` in Abbildung 8-7 dar. Sie beinhaltet neben einem links dargestellten Untermenü einen Platzhalter, in dem sie weitere Routen aktivieren kann.

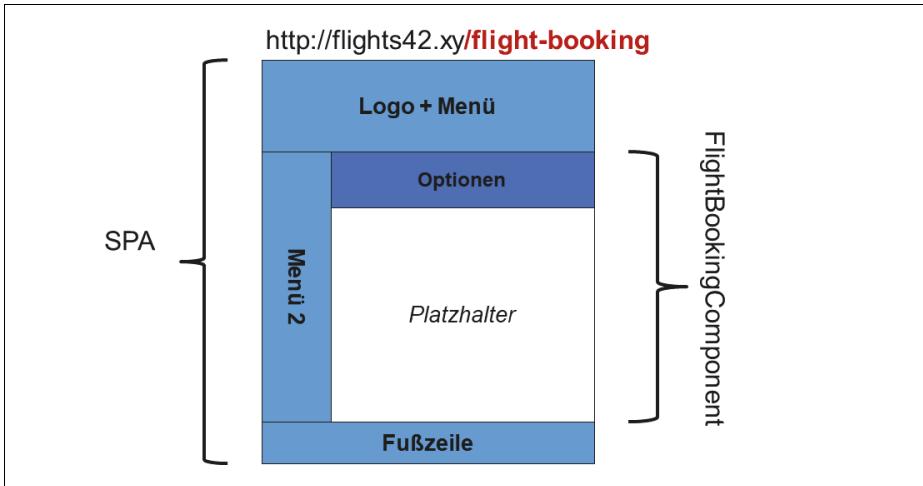


Abbildung 8-7: Komponente mit Child-Routes

Nun hängt es von der gewählten URL ab, welche Child-Route der Router im Platzhalter der betrachteten Komponente aktiviert. In Abbildung 8-8 gibt die verwendete URL beispielsweise an, dass die **FlightSearchComponent** innerhalb der **FlightBookingComponent** zu aktivieren ist.

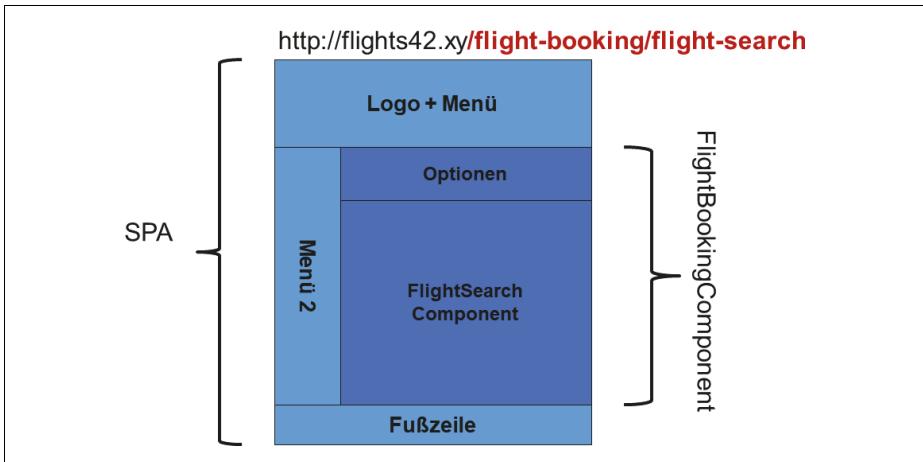


Abbildung 8-8: Hierarchisches Routing

## Child-Komponente implementieren

Um die Implementierung von Child-Routes zu veranschaulichen, fassen wir die Menüpunkte *Flights* und *Passengers* unserer Demoanwendung zu einem Menüpunkt *Flight Booking* zusammen (siehe Abbildung 8-9).

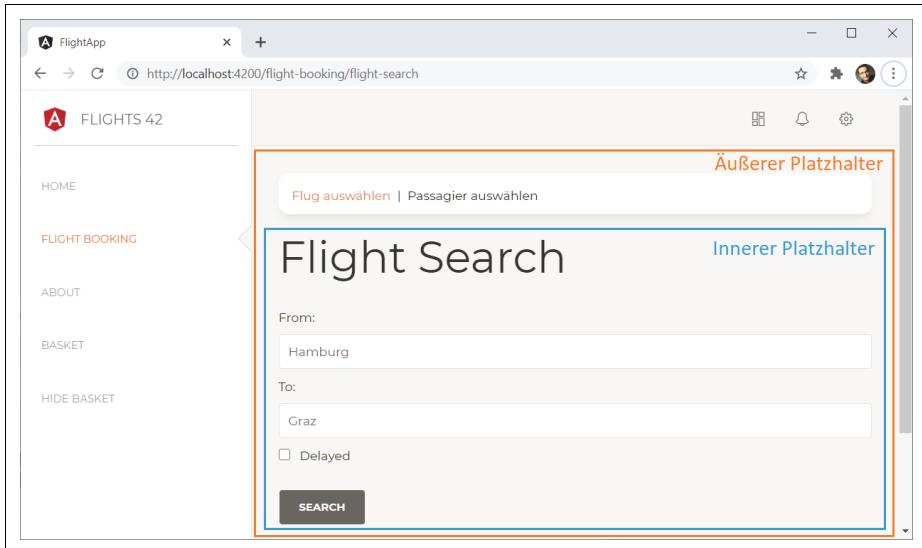


Abbildung 8-9: Beispielanwendung, erweitert um hierarchisches Routing

Die FlightBookingComponent können wir mit der Angular CLI generieren:

```
ng generate component flight-booking/flight-booking --flat
```

Bitte achten Sie auch hier darauf, dass diese Komponente innerhalb des Ordners *flight-booking* generiert und somit beim sich darin befindlichen FlightBooking Module registriert wird. Der Schalter `--flat` gibt an, dass kein weiterer Ordner generiert werden soll – der Ordner *flight-booking* existiert in unserem Fall ja schon.

Alternativ dazu können Sie auch das *Angular Schematics*-Plug-in in Visual Studio Code verwenden.

Das Template der neuen Komponente erhält ein `router-outlet` sowie Links, die auf die Routen `./flight-search` und `./passenger-search` verweisen (siehe Beispiel 8-11).

*Beispiel 8-11: Die FlightBookingComponent besteht aus zwei Links und einem Platzhalter.*

```
<div class="card">
  <div class="card-body">

    <ul class="nav nav-secondary">
      <li routerLinkActive="active">
        <a [routerLink]="/flight-search">Flug auswählen</a>
      </li>

      &nbsp; | &nbsp;
      <!-- Trennzeichen zwischen Menüpunkten -->

      <li routerLinkActive="active">
        <a [routerLink]="/passenger-search">Passagier auswählen</a>
      </li>
    </ul>
  </div>
</div>
```

```

        </li>
    </ul>
</div>
</div>

<router-outlet></router-outlet>
```



Mit dem Präfix `.` zeigt `routerLink` an, dass die definierte Route an die aktuelle Route anzuhängen ist. Da es sich dabei um das Standardverhalten handelt, könnte es diese Information auch weglassen. Interessant wird es jedoch, wenn ein Schwesterknoten zu adressieren ist: Mit `../passenger-search` können Sie zum Beispiel von `flight-booking/flight-search` zu `flight-booking/passenger-search` navigieren.

Für die Klasse `nav` bietet unser Theming Styles, die die Menüeinträge nebeneinander präsentieren. Allerdings bietet es keine Styles, um aktive Menüeinträge in dieser Konstellation hervorzuheben. Das müssen wir in der Datei `src/styles.scss` nachrüsten:

```

.nav-secondary li a {
    color: black;
}

.nav-secondary li.active a {
    color: #ef8157;
}
```

## Child-Komponente registrieren

Damit der Router unsere neue `FlightBookingComponent` aktivieren kann, ist sie in die Routenkonfiguration einzutragen (siehe Beispiel 8-12).

*Beispiel 8-12: Child-Routes in der Routing-Konfiguration für das FlightBookingModule*

```
// src/app/flight-booking/flight-booking.routes.ts
```

```

import { Routes } from '@angular/router';
import { FlightBookingComponent } from './flight-booking.component';
import { FlightSearchComponent } from './flight-search/flight-search.component';
import { PassengerSearchComponent } from './passenger-search/passenger-search.
component';

export const FLIGHT_BOOKING_ROUTES: Routes = [
{
    path: 'flight-booking',
    component: FlightBookingComponent,
    children: [
        {
            path: '',
            redirectTo: 'flight-search',
            pathMatch: 'full'
```

```

        },
        {
            path: 'flight-search',
            component: FlightSearchComponent
        },
        {
            path: 'passenger-search',
            component: PassengerSearchComponent
        },
        {
            path: 'flight-edit/:id',
            component: FlightEditComponent
        }
    ]
},
];

```

Die anderen Routen dieser Konfiguration werden *flight-booking* unter `children` zugeteilt. Somit kann der Router diese Child-Komponenten innerhalb des Platzhalters von *flight-booking* aktivieren.

Ein Aufruf von *flight-booking/flight-search* führt zum Beispiel dazu, dass die `FlightSearchComponent` innerhalb der `FlightBookingComponent` aktiviert wird. Diese erscheint wiederum innerhalb der `AppComponent`.

Beachten Sie bitte auch die Route ohne Namen. Dabei handelt es sich um die Standardroute innerhalb von *flight-booking*: Wird nur der Pfad *flight-booking* aufgerufen, erfolgt eine Umleitung auf *flight-booking/flight-search*.

## Hyperlinks zum Aktivieren von Child-Routen einrichten

Nun müssen wir noch einen Menüeintrag für die neue *flight-booking*-Route im Template der `SidebarComponent` einrichten (siehe Beispiel 8-13).

*Beispiel 8-13: Das Template der SidebarComponent verweist auf die Route flight-booking.*

```

<!-- src/app/sidebar/sidebar.component.html -->
[...]
<!-- Diesen Eintrag hinzufügen: -->
<li routerLinkActive="active">
    <a routerLink="flight-booking">
        <p>Flight Booking</p>
    </a>
</li>

<!-- Diese beiden Einträge entfernen: -->
<!-- <li routerLinkActive="active">
    <a routerLink="flight-search">
        <p>Flights</p>
    </a>
</li>

```

```

<li routerLinkActive="active">
  <a routerLink="passenger-search">
    <p>Passengers</p>
  </a>
</li> -->

```

Starten Sie nun die Anwendung (`ng serve -o`), können Sie über *Flight Booking* im Hauptmenü auf die `FlightBookingComponent` navigieren. Danach erhalten Sie die Möglichkeit, auf *Flights* oder auf *Passengers* zu wechseln (siehe Abbildung 8-10).

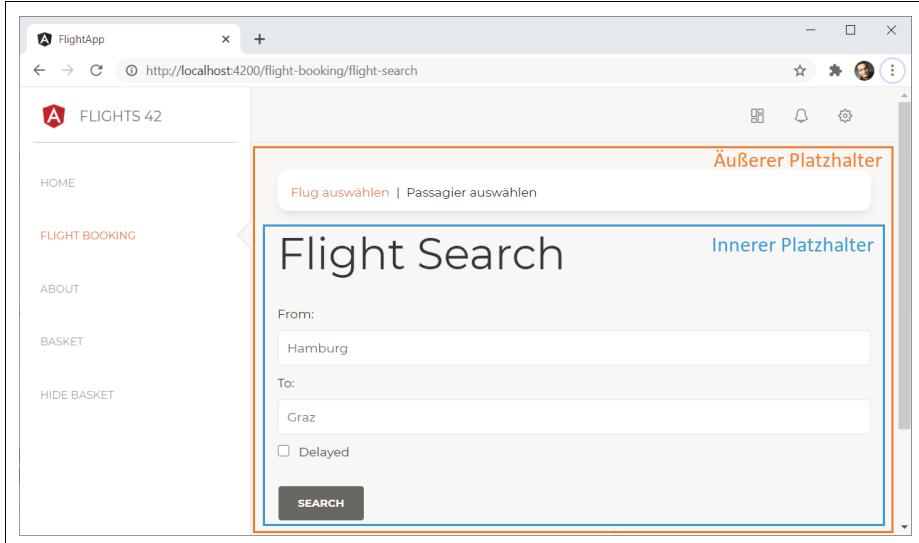


Abbildung 8-10: *FlightBookingComponent mit Child-Routes*

## Aux-Routes

Mit den als Aux-Routes bezeichneten *Auxiliary Routes* (Hilfsrouten) erlaubt der Router das Einführen mehrerer Platzhalter. Dabei müssen Sie jedem zusätzlichen Platzhalter einen Namen spendieren. Mit Angabe dieses Namens kann die Anwendung eine Komponente für den Platzhalter festlegen.

Das Konzept hinter Aux-Routes erlaubt es auch, alle Platzhalter unabhängig voneinander mit Inhalten zu füllen (siehe Abbildung 8-11).

Die Tatsache, dass in Abbildung 8-11 der Platzhalter mit dem Namen `aux` unter dem Standardplatzhalter positioniert ist, bedeutet übrigens nicht, dass sein Inhalt auch immer an dieser Stelle eingeblendet wird. Die Möglichkeiten von CSS erlauben es, die dort eingeblendeten Inhalte überall auf der Seite zu platzieren und sogar den Standardplatzhalter zu überblenden. Auf diese Weise lassen sich mit Aux-Routes modale Dialoge entwickeln, deren Zustand die URL widerspiegelt.

Weitere Anwendungsfälle sind das Einblenden zusätzlicher Informationen sowie die Realisierung einer Anwendung, die – wie eine Entwicklungsumgebung auch –

aus verschiedenen Bereichen besteht, die bis zu einem bestimmten Punkt sogar unabhängig voneinander sein können.

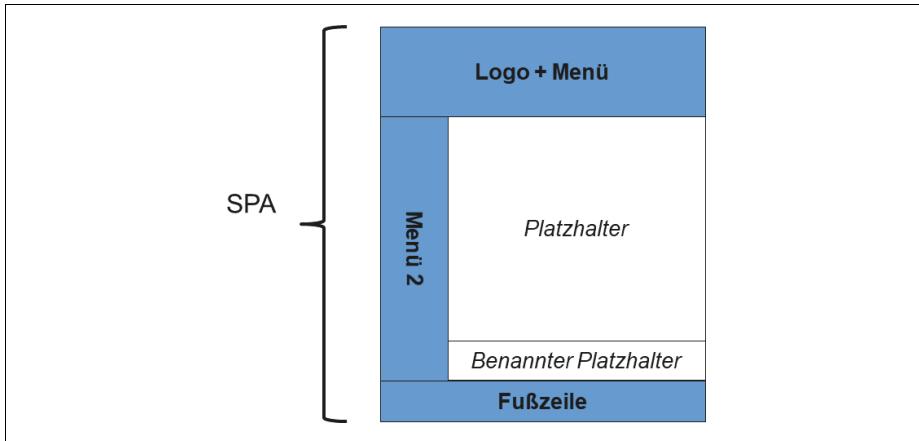


Abbildung 8-11: Schematische Darstellung von Aux-Routes

In diesem Abschnitt nutzen wir zur Veranschaulichung von Aux-Routes eine `BucketComponent`, die die ausgewählten Flüge listet. Bei Bedarf wird sie im Aux-Platzhalter aktiviert (siehe Abbildung 8-12).

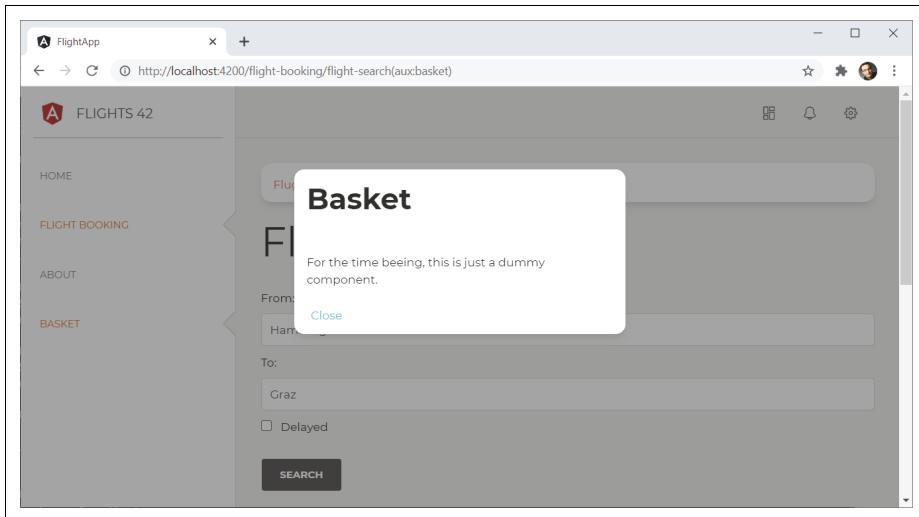


Abbildung 8-12: Nutzung einer Aux-Route in der Demoanwendung

## Platzhalter für Aux-Routes

Analog zum Standardplatzhalter erhält die `AppComponent` einen zweiten Platzhalter mit dem Namen `aux`:

```

<div class="content">
  <router-outlet></router-outlet>
  <!-- Benannten Platzhalter hinzufügen: -->
  <router-outlet name="aux"></router-outlet>
</div>

```

Der Name dieser router-outlets kann frei gewählt werden. Er muss also nicht auf aux lauten.

## Komponente für Aux-Route erzeugen

Die Komponente für die hier beschriebene Aux-Route wird wie gewohnt mit der CLI oder mit dem Visual-Studio-Code-Plug-in *Angular Schematics* generiert:

```
ng generate component basket
```

Die BasketComponent ist lediglich ein Dummy, das als Routing-Ziel dient. Für das Template können Sie das Markup in Beispiel 8-14 verwenden.

*Beispiel 8-14: Template der BasketComponent*

```
<!-- src/app/basket/basket.component.html -->

<div class="card-background"></div>

<div class="card-container">
  <div class="card">

    <div class="card-header">
      <h2 class="title">Basket</h2>
    </div>

    <div class="card-body">
      For the time being, this is just a dummy component.
    </div>

  </div>
</div>
```

Die Idee dabei ist, diese Komponente als Pop-over zu präsentieren. Das div mit der Klasse card-background soll deswegen halb transparent sein und sich über die gesamte Seite erstrecken. Der card-container hat eine Höhe und eine Breite von 100%. Darin zentrieren wir das div mit der Klasse card.

Um diese Überlegungen umzusetzen, verwenden wir die Stylings in Beispiel 8-15.

*Beispiel 8-15: Styling für Pop-over*

```
/* src/app/basket/basket.component.scss */

.card-background {
```

```

    opacity: 0.4;
    background-color: black;
    left: 0px;
    top: 0px;
    width: 100%;
    height: 100%;
    position: fixed;
    z-index: 10000;
}

.card-container {
    left: 0px;
    top: 0px;
    width: 100%;
    height: 100%;
    position: fixed;
    z-index: 11000;
}

.card {
    background-color: white;
    margin: 100px auto;
    width: 400px;
}

```

## Konfiguration für Aux-Route

Die Konfiguration von Aux-Routes entspricht prinzipiell jener von normalen Routen. Der einzige Unterschied besteht darin, dass sie mit der Eigenschaft outlet dem gewünschten router\_outlet zugewiesen wird (siehe Beispiel 8-16).

*Beispiel 8-16: Definition einer Aux-Route*

```
// src/app/app.routes.ts

[...]
// Import hinzufügen:
import { BasketComponent } from './basket/basket.component';

export const APP_ROUTES: Routes = [
    [...]
    // Eintrag hinzufügen:
    {
        path: 'basket',
        component: BasketComponent,
        outlet: 'aux'
            // Name des router-outlets
    },
    // Dieser Eintrag muss der letzte sein!
    {
        path: '**',
        component: NotFoundComponent
    }
];
```

Die Eigenschaft `outlet` muss mit dem Namen des `router-outlet` übereinstimmen. Genau genommen ist sogar jede Route mit einem bestimmten Platzhalter verbunden. Benennt eine Route ihren Platzhalter nicht per `outlet`, weist der Router ihr den Standardplatzhalter zu. Dieser nennt sich per definitionem `primary`.



Möchten Sie die gleiche Komponente in verschiedenen Platzhaltern aktivieren, müssen Sie pro Platzhalter einen eigenen Konfigurationseintrag einrichten. Da Routenkonfigurationen lediglich Arrays sind, lässt sich dieser Vorgang auch automatisieren.

## Verweise auf Aux-Routes einrichten

Die im Hauptplatzhalter zu aktivierende Route wird, wie üblich, durch die URL repräsentiert. Die Routen für Aux-Platzhalter werden innerhalb runder Klammern in der URL angeführt. Die URL

`http://localhost:4200/flight-booking/flight-search(aux:basket)`

weist beispielsweise darauf hin, dass die Route `flight-booking/flight-search` im Hauptplatzhalter zu aktivieren ist, sowie darauf, dass die Route `basket` dem Platzhalter mit dem Namen `aux` eine Komponente verpassen soll.

Diese Schreibweise mag ungewohnt wirken. Trotzdem handelt es sich dabei um eine standardkonforme URL.



Um mehrere Aux-Platzhalter gleichzeitig zu bestücken, nutzen Sie das Trennzeichen `//`: `http://localhost:4200/flight-booking/flight-search(aux:basket//otherOutlet:otherRoute)`

Auch solche URLs lassen sich mit `routerLink` erzeugen:

```
<!-- src/app/sidebar/sidebar.component.html -->
[...]
<li routerLinkActive="active">
  <a [routerLink]="[{outlets: {aux: 'basket'}}]">
    <p>Basket</p>
  </a>
</li>
[...]
```

Während die Direktive `routerLink` in den vorangegangenen Abschnitten lediglich auf die Route für den Standardplatzhalter verwiesen hat, definiert sie hier ein Objekt, das mit der Eigenschaft `outlet` auf ein weiteres Objekt verweist. Letzteres bildet die Namen der Platzhalter auf die dafür zu aktivierenden Routen ab. Auf diese Weise lassen sich auch Komponenten für mehrere Platzhalter gleichzeitig aktivieren.

Ein Aux-Platzhalter muss nicht zwingend eine aktive Route haben. Um einen Platzhalter zu leeren, müssen Sie ihm lediglich den Wert `null` zuweisen. Damit lässt sich

zum Beispiel eine Schaltfläche zum Schließen des Pop-overs im Template der Basket Component anbieten:

```
<!-- src/app/basket/basket.component.html -->
[...]

<div class="card-footer">
  <a [routerLink]="/" {outlets: {aux: null}}>Close</a>
</div>
```

Bitte beachten Sie den Array-Eintrag `/`: Dieser ist hier wichtig, da Angular Pfade standardmäßig immer relativ zur aktuellen Route interpretiert. Die aktuelle Route ist im Fall der BasketComponent jedoch die hier besprochene Aux-Route. Deswegen sorgt der Eintrag `/` dafür, dass dieser Pfad als absoluter Pfad interpretiert wird.

Die Eigenschaften im Objekt `outlets` nehmen auch Arrays zum Beschreiben von Routen entgegen. Damit lassen sich Parameter an Aux-Routes übergeben:

```
<div class="card-footer">
  <a [routerLink]="/" {outlets: {aux: ['basket', 17]}}>Close</a>
</div>
```

Um die Routen in mehreren Platzhaltern gleichzeitig zu wechseln, müssen Sie lediglich mehrere Eigenschaften im `outlets`-Objekt platzieren:

```
<a [routerLink]=["{outlets: {primary: 'flight-booking/flight-search',
aux: 'basket'}}"]>
  <p>Basket</p>
</a>
```

Da `primary` auch der Name des Hauptplatzhalters ist, lässt sich dieser spezielle Fall darüber hinaus so ausdrücken:

```
<a [routerLink]=["['flight-booking/flight-search', {outlets: {aux: 'basket'}}]"]>
  <p>Basket</p>
</a>
```

## Mit dem Query-String und dem Hash-Fragment arbeiten

Standardmäßig stellt der Router die übergebenen Parameter als Matrixparameter innerhalb der URL dar. Wie im letzten Abschnitt gezeigt, beziehen sich diese immer auf ein URL-Segment und somit auf eine Komponente. Zusätzlich kann die Anwendung jedoch auch einen klassischen Query-String in der bekannten Form

```
url?param1=value1&param2=value2&param3=value3
```

definieren. Die Definition eines Hash-Fragments ist ebenfalls möglich:

```
url#info-im-hashfragment
```

Diese beiden Optionen bieten sich an, um anwendungsweite Einstellungen zu repräsentieren.

In diesem Abschnitt gehen wir auf diese Möglichkeiten ein. Um unser Beispiel nicht zu überfrachten, setzen nachfolgende Kapitel nicht auf den hier besprochenen Anpassungen auf. Falls Sie die Demoanwendung nachbauen, sollten Sie deswegen an dieser Stelle eine Sicherung davon machen oder für diesen Abschnitt einen eigenen Branch in Ihrer Quellcodeverwaltung verwenden.

## QueryString und Hash-Fragment programmatisch beeinflussen

Um den QueryString und das Hash-Fragment programmatisch zu verändern, nimmt die `navigate`-Methode des Routers über ihr optionales zweites Argument ein `NavigationExtra`-Objekt entgegen. Ihre Eigenschaft `queryParams` verweist auf ein Objekt, das die gewünschten URL-Parameter widerspiegelt (siehe Beispiel 8-17).

*Beispiel 8-17: QueryString beim programmgesteuerten Navigieren setzen*

```
// src/app/navbar/navbar.component.ts

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

// Importe hinzufügen:
import { NavigationExtras, Router } from '@angular/router';

@Component( [...] )
export class NavbarComponent {

    [...]
    expertMode = false;

    constructor(private router: Router) {
    }

    [...]

    // Methode hinzufügen:
    activateExpertMode() {
        this.expertMode = !this.expertMode;

        const extras: NavigationExtras = {
            queryParams: {
                expertMode: this.expertMode
            }
        };

        this.router.navigate([], extras);
    }
}
```

Zum Ausprobieren können Sie diese Methode an einen neuen Menüeintrag im Template der `NavbarComponent` binden (siehe Beispiel 8-18).

*Beispiel 8-18: Menüeintrag in der NavbarComponent*

```
<!-- src/app/navbar/navbar.component.html -->
[...]
<ul class="navbar-nav">
  <li class="nav-item">
    <a class="nav-link btn-magnify" (click)="activateExpertMode()">
      Expert Mode
    </a>
  </li>
  [...]
</ul>
```

Neben der Eigenschaft `queryParams` kann der Aufrufer auf diese Weise noch weitere Informationen übergeben (siehe Tabelle 8-2).

*Tabelle 8-2: Ausgewählte Eigenschaften von NavigationExtra*

Eigenschaft	Beschreibung
<code>queryParams</code>	Definiert ein Objekt mit den zu setzenden URL-Parametern.
<code>queryParamsHandling</code>	Nimmt zwei Werte an: <code>preserve</code> legt fest, dass der bestehende Query-String beim aktuellen Routenwechsel erhalten bleibt, und <code>merge</code> erweitert den aktuellen Query-String um die in <code>queryParams</code> angegebenen Parameter.
<code>hash</code>	Definiert einen String als Hash-Fragment.
<code>preserveHash</code>	Definiert, dass der Router das aktuelle Hash-Fragment im Rahmen der angeforderten Navigation erhalten soll.

Beispiel 8-19 zeigt ein Beispiel für die Nutzung dieser Eigenschaften.

*Beispiel 8-19: Die NavigationExtras erlauben das Setzen von Query-String und Hash-Fragment.*

```
// src/app/navbar/navbar.component.ts
```

```
import { Component } from '@angular/core';
import { NavigationExtras, Router } from '@angular/router';

@Component( [...] )
export class NavbarComponent {

  [...]
  expertMode = false;
  experimental = false;
  darkMode = false;

  constructor(private router: Router) {
  }

  [...]

  // Methode aktualisieren:
  activateExpertMode() {
    this.expertMode = !this.expertMode;
  }
}
```

```

        const extras: NavigationExtras = {
            queryParams: {
                expertMode: this.expertMode
            },
            queryParamsHandling: 'merge',
            preserveFragment: true
        };

        this.router.navigate([], extras);
    }

    // Methode hinzufügen:
    activateExperimentalFeatures() {
        this.experimental = !this.experimental;

        const extras: NavigationExtras = {
            queryParams: {
                experimental: this.experimental
            },
            queryParamsHandling: 'merge',
            preserveFragment: true
        };

        this.router.navigate([], extras);
    }

    // Methode hinzufügen:
    activateExpertModeInHash() {

        this.darkMode = !this.darkMode;

        const extras: NavigationExtras = {
            fragment: this.darkMode ? 'dark-mode' : 'light-mode',
            queryParamsHandling: 'preserve'
        };

        this.router.navigate([], extras);
    }
}

```

Die Methoden `activateExpertMode` und `activateExperimentalFeatures` aktualisieren den Query-String, erhalten aber bestehende Eigenschaften sowie das Hash-Fragment. Die Methode `activateExpertModeInHash` aktualisiert das Hash-Fragment und erhält auch den Query-String.

Um diese Methoden auszuprobieren, können Sie analog zu Beispiel 8-18 weitere Menüeinträge einführen.

## Query-String und Hash-Fragment deklarativ beeinflussen

Erfolgt das Routing nicht programmgesteuert über die Methode `navigate`, sondern deklarativ über die Direktive `routerLink`, lassen sich die im letzten Abschnitt diskutierten Optionen per Datenbindung angeben (siehe Beispiel 8-20).

*Beispiel 8-20: Deklaratives Verändern von Query-String und Hash-Fragment*

```
<!-- src/app/sidebar/sidebar.component.html -->

[...]
<li routerLinkActive="active">
  <a href="#" routerLink="home" [queryParams]="{{more: 42}}"
    queryParamsHandling="merge" [preserveFragment]="true">
    <p>Home</p>
  </a>
</li>

<li routerLinkActive="active">
  <a routerLink="flight-search" [queryParams]="{{more: 43}}"
    queryParamsHandling="merge" [preserveFragment]="true">
    <p>Flights</p>
  </a>
</li>

<li routerLinkActive="active">
  <a routerLink="passenger-search" [queryParams]="{{more: 44}}"
    queryParamsHandling="merge" fragment="light-mode">
    <p>Passengers</p>
  </a>
</li>

<li routerLinkActive="active">
  <a routerLink="about" [queryParams]="{{more: 45}}"
    queryParamsHandling="merge" [preserveFragment]="true">
    <p>About</p>
  </a>
</li>
[...]
```

## Query-String und Hash-Fragment auslesen

Zum Auslesen des Query-Strings bietet die ActivatedRoute ein Observable query Params. Für das Auslesen des Hash-Fragments bietet sie analog dazu ein Observable fragment (siehe Beispiel 8-21).

*Beispiel 8-21: Query-String und Hash-Fragment auslesen*

```
// src/app/flight-booking/flight-edit/flight-edit.component.ts

[...]

@Component( [...] )
```

```

export class FlightEditComponent implements OnInit {

  [...]

  constructor(private route: ActivatedRoute) { }

  ngOnInit(): void {
    [...]

    this.route.queryParams.subscribe(p => {
      console.debug('queryParams', p);
    });

    this.route.fragment.subscribe(p => {
      console.debug('fragment', p);
    });
  }
}

```

## HTML5-Routing vs. Hash-Routing

Bis jetzt haben wir den Pfad der aktiven Route einfach an die URL unserer SPA angehängt:

`http://localhost:4200/flight-booking/flight-search`

Diese Vorgehensweise nennt sich landläufig auch HTML5-Routing, weil sie auf Erweiterungen basiert, die mit HTML5 eingeführt wurden. Daneben unterstützt Angular auch noch das sogenannte Hash-basierte Routing, das vor allem bei älteren SPAs üblich war. Hier befindet sich die Route im Hash-Fragment der URL:

`http://localhost:4200#/flight-booking/flight-search`

Um flexibel zu bleiben, überlässt der Router die Verwaltung von URLs einer austauschbaren Strategieimplementierung. Die standardmäßig verwendete `PathLocationStrategy` nutzt HTML5-Routing, während die `HashLocationStrategy` auf Hash-basiertes Routing setzt.

Dieser Abschnitt erläutert, wie Sie die gewünschte Strategie festlegen und welche Konsequenzen damit einhergehen.

### PathLocationStrategy

Eine Herausforderung beim Einsatz der `PathLocationStrategy` besteht darin, zu erkennen, welcher Teil der URL serverseitig und welcher in der SPA als Route zu interpretieren ist.

Serverseitig löst man dieses Problem durch Konfiguration des jeweiligen Webservers. Gehen wir davon aus, dass wir unsere Anwendung unter `https://www.flights42.xy` veröffentlichen, müsste der Webserver jede Anfrage, die mit dieser URL beginnt, auf unsere `index.html` umleiten.

Der von `ng serve` gestartete Webserver macht das übrigens ähnlich. Allerdings leitet er alle Anfragen für URLs, die er nicht kennt, an `http://localhost:4200` weiter. Sehen Sie bitte 4200 stellvertretend für den verwendeten Port.

Clientseitig lässt sich dieses Problem mit dem `base`-Element lösen, das die Angular CLI in der `index.html` beim Erzeugen neuer Projekte hinterlegt:

```
<head>
  [...]
  <base href="/">
  [...]
</head>
```

Dieses verweist auf jenen Teil der URL, der serverseitig zu interpretieren ist. Den Rest wertet Angular aus. Außerdem hilft es dem Browser, die Pfade, die zu statischen Assets wie Bildern oder Schriftarten führen, korrekt aufzulösen.

Die Standardeinstellung `href="/"` geht davon aus, dass Ihre Anwendung über das Hauptverzeichnis Ihrer Domäne erreichbar ist. Bei unserem Gedankenexperiment wäre das `https://www.flights42.xy`.

Stellen Sie Ihre Anwendung jedoch in einem Unterverzeichnis bereit, ist dieses Unterverzeichnis unter `href` einzutragen. Nehmen wir zur Veranschaulichung das Unterverzeichnis `flight-app` an. Die gesamte URL, die zur Anwendung führt, wäre nun `https://www.flights42.xy/flight-app`, und der `base`-Eintrag würde auch auf `flight-app` verweisen:

```
<head>
  [...]
  <base href="/flight-app">
  [...]
</head>
```

Das manuelle Ändern dieses Eintrags ist natürlich lästig und fehleranfällig. Deswegen bietet `ng build` den Schalter `--base-href` an, mit dem Sie den gewünschten Pfad beim Build angeben können:

```
ng build --base-href flight-app
```

Die CLI platziert nun diesen Wert im `href`-Attribut. Allerdings wirkt sich diese Einstellung nur auf die erzeugte Anwendung aus, die sich nach dem Built im `dist`-Ordner befindet.

Alternativ zur Nutzung des `base`-Tags lässt sich das Unterverzeichnis der Anwendung auch per Dependency Injection angeben. Dazu platzieren Sie in Ihrem App Module einen Serviceprovider für das Token `APP_BASE_HREF`:

```
[...]
import {APP_BASE_HREF} from '@angular/common';

@NgModule({
  [...]
  providers: [{provide: APP_BASE_HREF, useValue: '/flight-app'}]
})
class AppModule {}
```

Aufgrund dieses Providers weiß zwar Angular, welcher Teil der URL für das Routing zu nutzen ist, der Browser weiß jedoch nicht, wie er Pfade, die zu statischen Assets führen, auflösen soll. Darum müssen Sie sich selbst kümmern, z.B. indem Sie lediglich absolute Pfade verwenden.

## HashLocationStrategy

Die HashLocationStrategy lässt sich beim Einrichten der Routen für das AppModule aktivieren:

```
// src/app/app.module.ts

[...]

@NgModule({
  imports: [
    RouterModule.forRoot(APP_ROUTES, { useHash: true }),
    [...]
  ],
  [...]
})
export class AppModule { }
```

Die Methode `forRoot` nimmt neben der Routenkonfiguration ein weiteres Objekt entgegen. Um die HashLocationStrategy zu erhalten, setzen Sie dessen Eigenschaft `useHash` auf `true`.

In diesem Fall müssen Sie weder serverseitig eine Umleitung konfigurieren noch mit dem `base`-Element in der `index.html` den Ordner der SPA angeben. Die Trennung zwischen jenem Teil der URL, die serverseitig zu interpretieren ist, und der clientseitigen Route wird nun schließlich durch das Hash-Symbol in der URL angegeben.

Ein Nachteil dieser Strategie ist gegebenenfalls, dass die entstehende URL auf den Benutzer nicht ganz so natürlich wirkt. Außerdem funktioniert beim Einsatz dieser Strategie ein eventuelles serverseitiges Vorrendern der Anwendung nicht. Während Businessanwendungen diese Option in der Regel nicht nutzen, ist sie für öffentliche Websites wichtig, zumal sich durch das Vorrendern die (gefühlte) Startperformance verbessern lässt.

## Zusammenfassung

Der von Angular angebotene Router ermöglicht es, unterschiedliche Seiten innerhalb einer *Single Page Application* (SPA) zu simulieren. Um ihn zu nutzen, bilden Sie Pfade auf Komponenten ab. Finden sich diese Pfade in der aufgerufenen URL, aktiviert der Router die damit assoziierten Komponenten in einem Platzhalter der Seite. Außerdem können Sie über die URL Parameter an die aktivierte Komponente weitergeben.

Aktivierte Komponenten können wiederum Platzhalter aufweisen. Auf diese Weise lassen sich hierarchische Navigationsstrukturen realisieren. Aux-Routes erlauben daneben den Einsatz verschiedener Platzhalter, die sich unabhängig voneinander mit Komponenten bestücken lassen.

# Template-getriebene Formulare und Validierung

Für die Verwaltung von Formularen bringt Angular gleich zwei Ansätze mit: einen *Template-getriebenen* und einen *reaktiven*. Ersterer sieht vor, dass der Entwickler die benötigten Formulare zur Gänze über HTML-Markup beschreibt. Entscheidet sich der Entwickler hingegen für den reaktiven Ansatz, beschreibt er das Formular über einen Objektgraphen. Namensgebend ist hier die Tatsache, dass dieser Objektgraph einen direkten Zugriff auf Observables bietet, die aus der Welt von RxJS (*Reactive Extensions for JavaScript*) stammen und die über Zustandsänderungen an den Eingabefeldern informieren. Während der Template-getriebene Ansatz in der Regel mit weniger Aufwand einhergeht, bietet die reaktive Alternative dem Entwickler mehr Kontrolle.

Dieses Kapitel stellt die Template-getriebenen Formulare vor. Dazu erweitert es das Suchformular unserer Demoanwendung um Validierungsregeln (siehe Abbildung 9-1).

The screenshot shows a 'Flight Search' form with the following fields and errors:

- From:** A text input field containing 'D' which has a red border, indicating a validation error. Below it, red text says: "Es liegen Validierungsfehler für diese Eingabe vor." and "Dieses Feld ist ein Pflichtfeld."
- To:** A text input field containing 'G' which has a green border, indicating no validation error.
- Delayed:** A checkbox labeled "Delayed" with an unchecked state.

At the bottom is a dark grey button labeled "SEARCH".

Abbildung 9-1: Suchformular mit Validierung

Das nächste Kapitel bespricht darauf aufbauend die reaktiven Formulare.

# FormsModule einbinden

Angular ist sehr modular aufgebaut. Das geht sogar so weit, dass die Funktionalität für Template-getriebene Formulare in einem eigenen Modul zu finden ist. Dieses nennt sich `FormsModule` und befindet sich (wie die meisten in diesem Kapitel besprochenen Framework-Bestandteile) im Namensraum `@angular/forms`. Dieses Modul müssen Sie in jedes eigene Modul einbinden, das Template-getriebene Formulare benötigt.

Da unsere `FlightSearchComponent` für die Datenbindung die Direktive `ngModel` aus der Welt der Template-getriebenen Formulare verwendet, haben wir das `FormsModule` bereits im `FlightBookingModule` eingebunden (siehe Beispiel 9-1).

*Beispiel 9-1: Das FlightBookingModule referenziert das FormsModule.*

```
// src/app/flight-booking/flight-booking.module.ts

[...]
import { FormsModule } from '@angular/forms';

@NgModule({
  imports: [
    RouterModule.forChild(FLIGHT_BOOKING_ROUTES),
    // Hier wurde das FormsModule importiert:
    FormsModule,
    SharedModule
  ],
  declarations: [
    [...]
  ],
  exports: [
    [...]
  ]
})
export class FlightBookingModule { }
```

## Eingaben validieren

Steht das `FormsModule` erst einmal zur Verfügung, können Templates einzelne Eingabesteuerelemente mittels `ngModel` und Two-Way-Data-Binding an die jeweiligen Eigenschaften binden:

```
<input [(ngModel)]="from" name="from" class="form-control">
```

Kommt das `input`-Element innerhalb eines `form`-Elements zum Einsatz, erzwingt Angular wie hier gezeigt die Nutzung des `name`-Attributs. Damit benennt Angular die Objekte im erwähnten Objektgraphen.



Spinden Sie sämtlichen mit `ngModel` gebundenen Fehlern in Ihrer `FlightSearchComponent` ein eindeutiges `name`-Attribut.

Zusätzlich besteht die Möglichkeit, Validierungsregeln über Attribute anzugeben. Das folgende Beispiel zeichnet auf diese Weise das Eingabefeld als Pflichtfeld mit mindestens drei Zeichen aus:

```
<input [(ngModel)]="from" name="from" class="form-control"
required minlength="3">
```

Tabelle 9-1 gibt Ihnen einen Überblick über die Validierungsregeln, die Angular ab Werk liefert.

Tabelle 9-1: Built-in-Validatoren

Attribut	Validierung
required	Pflichtfeld.
minlength	Minimale String-Länge.
maxlength	Maximale String-Länge.
email	Validiert eine E-Mail-Adresse gegen den typischen Aufbau einer E-Mail-Adresse.
pattern	Prüfung gegen regulären Ausdruck.
min	Validiert eine number gegen eine untere Schranke.
max	Validiert eine number gegen eine obere Schranke.
requireTrue	Validiert, ob ein Wert true (bzw. <i>truthy</i> ) ist. Sie verwenden diesen Validator, um zu prüfen, ob eine Checkbox aktiviert wurde.

Wenn Sie mit diesen von Angular angebotenen Validierungsregeln nicht auskommen, können Sie auch – wie im Abschnitt »Eigene Validierungsdirektiven« auf Seite 212 beschrieben – eigene Validierungsregeln implementieren.

## Zugriff auf den Zustand des Formulars

Für Template-getriebene Formulare erzeugt Angular im Hintergrund einen Objektgraphen, der seinen Zustand beschreibt (siehe Abbildung 9-2).

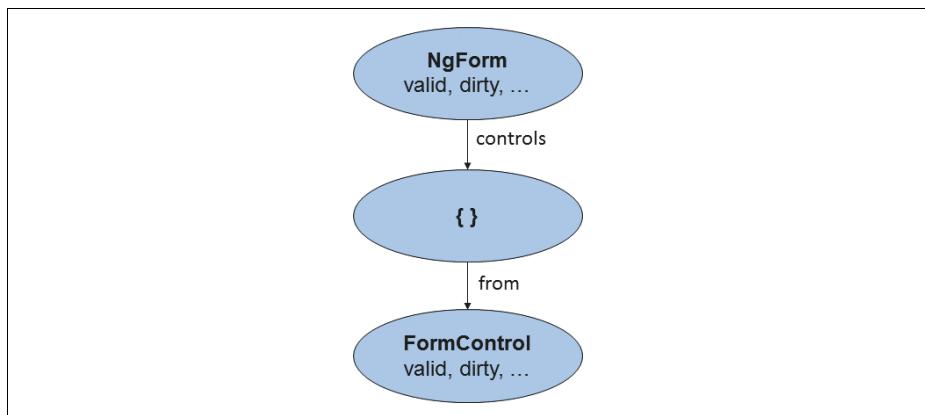


Abbildung 9-2: Objektgraph zur Beschreibung eines Formulars

An der Spitze dieses Graphen finden Sie ein Objekt vom Typ `NgForm`. Es handelt sich dabei um eine Angular-Direktive. Sie steht für das gesamte Formular und zeigt unter anderem an, ob es korrekt validiert (*valid*) oder verändert (*dirty*) wurde. Nur wenn sämtliche Felder des Formulars korrekt validiert wurden, sieht Angular das Formular als valide an. Wurde auch nur ein Feld verändert, sieht Angular das Formular hingegen als verändert an.

Zusätzlich enthält der Objektgraph pro Eingabefeld ein Objekt vom Typ `FormControl`. Diese Objekte lassen sich über die Auflistung `controls` von `NgForm` erreichen. Hierbei handelt es sich um ein Objekt, das als Dictionary verwendet wird und die Namen der Eingabefelder auf `FormControl`-Objekte abbildet. Diese Objekte spiegeln den Zustand der einzelnen Eingabefelder wider und geben ebenfalls Auskunft über den Ausgang der Validierung (*valid*) sowie darüber, ob das Feld eine Änderung erfahren hat (*dirty*).

Um eine Referenz auf diese Objekte zu erhalten, führen Sie eine sogenannte Template-Variable für das Formular ein. Solche Variablen verweisen auf eine Direktive oder eine Komponente, die hinter einem Element im Template steht. Beispiel 9-2 deklariert z. B. für das `form`-Element eine Template-Variable `f`.

*Beispiel 9-2: Template-Variable für ngForm*

```
<!-- src/app/flight-search/flight-search.component.html -->

<h1>Flight Search</h1>

<!-- Ergänzen Sie das form-Tag und weisen -->
<!-- Sie ihm eine Template-Variable zu: -->
<form #f="ngForm">

  <div class="form-group">
    <label>From:</label>

    <!-- Ergänzen Sie name="from" -->
    <input [(ngModel)]="from" name="from" class="form-control" required
           minlength="3">
  </div>

  <div class="form-group">
    <label>To:</label>

    <!-- Ergänzen Sie name="to" -->
    <input [(ngModel)]="to" name="to" class="form-control">
  </div>

  [...]

  <div class="form-group">
    <button [...]>Search</button>
  </div>
</form>
```

Im Zuge der Deklaration ist dem Handle eine Raute (#) voranzustellen. Da hinter einem HTML-Element eine Komponente sowie mehrere Direktiven stehen können, weist das betrachtete Beispiel dem Handle #f den Wert `ngForm` zu. Damit referenziert es die `NgForm`-Direktive. Dies funktioniert, da das Angular-Team `NgForm` unter diesem Namen veröffentlicht.

Das erste `input`-Element bindet das Beispiel an die Eigenschaft `from`, die sich im Komponentencontroller befindet, und weist die Validierungsattribute `required` und `minlength` auf. Wie schon erwähnt, kommt dem `name`-Attribut eine wichtige Eigenschaft zu: Es legt den Namen des `FormControl`-Objekts im Objektgraphen fest.

Auf dieses `FormControl`-Objekt greifen Sie über `f?.controls?.from` zu. Beispiel 9-3 nutzt das `FormControl` vom Eingabefeld für `from`, um den Validierungsstatus zu ermitteln und um gegebenenfalls einen Validierungsfehler auszugeben.

*Beispiel 9-3: Fehlermeldungen unter Nutzung von FormControl-Objekten ausgeben*

```
<!-- src/app/flight-search/flight-search.component.html -->

<h1>Flight Search</h1>

<form #f="ngForm">
  <div class="form-group">
    <label>From:</label>
    <input [(ngModel)]="from" name="from" required
           minlength="3" class="form-control">
  </div>

  <div class="error" *ngIf="!f?.controls?.from?.valid">
    Es liegen Validierungsfehler für diese Eingabe vor.
  </div>

  <div class="error" *ngIf="f?.controls?.from?.hasError('required')">
    Dieses Feld ist ein Pflichtfeld.
  </div>

  <div class="error" *ngIf="f?.controls?.from?.hasError('minlength')">
    Erfassen Sie bitte min. 3 Zeichen.
  </div>

  <div class="error" *ngIf="f?.controls?.from?.errors">
    Internes Errors-Objekt: {{ f?.controls?.from?.errors | json}}
  </div>

  [...]
</form>
[...]
```

Der Objektgraph steht leider nicht von Anfang an zur Verfügung, sondern mit einer kleinen Verzögerung. Das liegt daran, dass Angular zuerst für alle Elemente entsprechende Direktiven einrichten muss. Diese kurze Verzögerung wird wohl vom Benutzer kaum bemerkt, führt aber dazu, dass für einen kurzen Augenblick

`undefined`-Werte vorliegen. Deswegen nutzt dieses Beispiel zusätzlich den Safe-Access-Operator, der durch die Kombination aus einem Fragezeichen und einem Punkt gebildet wird: `f?.controls?.from`. Das vorangestellte Fragezeichen bewirkt, dass Angular den gesamten Ausdruck als `undefined` auswertet, wenn der Teil links davon `undefined` ist. Auf diese Weise verhindert es die Navigation über `undefined` hinweg, was zu einem Laufzeitfehler führen würde.



Auch für Eingabeelemente lassen sich Template-Variablen einführen. Damit können Sie die jeweiligen `FormControl`s direkt adressieren:

```
<input #from="ngModel"
       class="form-control"
       [(ngModel)]="from"
       required
       minlength="3"
       name="from">

<div *ngIf="from?.valid" class="error">
    Validierungsfehler!
</div>
```

Das gezeigte Beispiel prüft mittels `*ngIf` und der Eigenschaft `valid`, ob für `from` mindestens ein Validierungsfehler vorliegt. In diesem Fall gibt es eine Fehlermeldung aus. Zusätzlich prüft es, welcher Validierungsfehler vorliegt. Dazu kommt die Methode `hasError` zum Einsatz. Auf diese Weise bietet das Beispiel eine zusätzliche Information, die zu den aufgetretenen Fehlern passt.

Außerdem gibt das Beispiel das interne `errors`-Objekt aus. Dabei handelt es sich um ein Objekt, das für jeden aufgetretenen Fehler eine Eigenschaft mit weiteren Informationen besitzt. Darin findet sich zum Beispiel für `minlength`, wie viele Zeichen erwartet wurden. Somit könnte man diese Information in den Fehlermeldungen, die hier noch hartcodiert ist, dynamisch gestalten:

```
<div class="error" *ngIf="f?.controls?.from?.hasError('minlength')">
    Erfassen Sie bitte min.
    {{ f?.controls?.from?.errors?.minlength?.requiredLength }} Zeichen.
</div>
```

Um herauszufinden, für welche Validierungsregel `errors` welche Informationen bereitstellt, ist es sinnvoll, sie während der Entwicklung mit der JSON-Pipe auszugeben.

Damit diese Fehlermeldungen auch als solche wahrgenommen werden, empfiehlt sich das Einrichten eines Styles in der Datei `src/styles.scss`:

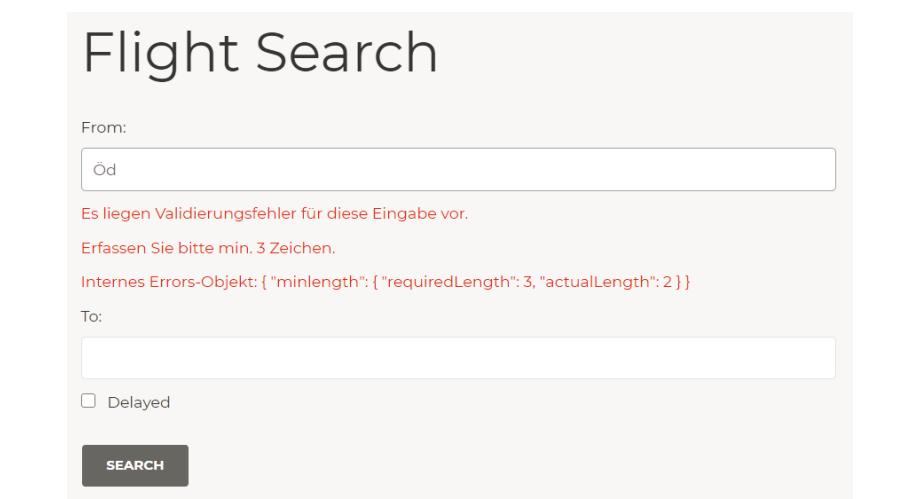
```
.error {
    color: red;
    margin-bottom: 10px;
}
```

Zusätzlich bietet es sich an, die `disabled`-Eigenschaft der Schaltfläche `Search` an den Validierungsstatus des gesamten Formulars zu binden:

```
<button
  class="btn btn-default"
  (click)="search()"
  [disabled]="!f?.valid">
  Search
</button>
```

Liegt im Formular ein Fehler vor, lässt sich die Schaltfläche nicht mehr betätigen.

Wenn Sie nun die Anwendung starten, sollten Sie sehen, dass die konfigurierten Validierungsregeln aktiv werden (siehe Abbildung 9-3).



The screenshot shows a 'Flight Search' application interface. At the top, there's a heading 'Flight Search'. Below it, there are two input fields: 'From:' containing 'Öd' and 'To:' which is empty. Under the 'From:' field, there is an error message: 'Es liegen Validierungsfehler für diese Eingabe vor.' followed by 'Erfassen Sie bitte min. 3 Zeichen.' and the internal error object 'Internes Errors-Objekt: { "minlength": { "requiredLength": 3, "actualLength": 2 } }'. There is also a checkbox labeled 'Delayed' and a 'SEARCH' button at the bottom.

Abbildung 9-3: Validierungsfehler von Built-in-Validatoren



Standardmäßig validiert Angular bei jeder Änderung der Eingabefelder. Ist Ihnen das zu viel, können Sie dieses Verhalten mit der Eigenschaft `ngModelOptions` anpassen:

```
<input
  [(ngModel)]="from"
  [ngModelOptions]="{ updateOn: 'blur' }"
  [...]>
```

Die Eigenschaft `updateOn` definiert, wann `ngModel` die vom Benutzer erfassten Werte in die gebundenen Eigenschaften validieren und zurückschreiben soll. Sie bietet die folgenden Optionen:

*change*

Die Eingaben werden bei jeder Änderung validiert und zurückgeschrieben.

*blur*

Die Eingaben werden beim Verlassen des Felds validiert und zurückgeschrieben.

*submit*

Die Eigenschaften werden beim Betätigen einer Schaltfläche im Formular validiert und zurückgeschrieben.

Falls Sie die Option `submit` ausprobieren möchten, sollten Sie zuvor die Bindung an `disabled` entfernen, um ein Henne-Ei-Problem zu vermeiden:

```
<button [...] [disabled]="f?.valid">Search</button>
```

## Bedingte Formatierung von Eingabefeldern

Abhängig vom aktuellen Zustand weist Angular den Eingabefeldern unterschiedliche Klassen zu. Diese kann eine Anwendung für bedingte Formatierungen nutzen, indem sie für diese Klassen Styles bereitstellt (siehe Tabelle 9-2).

Tabelle 9-2: Klassen für Formularfelder

Klasse	Zustand
<code>ng-invalid</code>	Das Feld wurde nicht korrekt validiert.
<code>ng-valid</code>	Für das Feld liegt kein Validierungsfehler vor.
<code>ng-dirty</code>	Das Feld wurde vom Benutzer verändert.
<code>ng-pristine</code>	Das Feld wurde vom Benutzer nicht verändert.
<code>ng-pending</code>	Die Validierung ist noch nicht abgeschlossen (siehe den Abschnitt »Asynchrone Validatoren« auf Seite 223).

Um nun Eingabefeldern mit fehlerhaften Eingaben einen roten Rand und Eingabefeldern mit korrekten Eingaben einen grünen Rand zu verpassen, kann die Anwendung das Stylesheet aus Beispiel 9-4 verwenden:

Beispiel 9-4: Styles für die bedingte Formatierung

```
/* src/styles.scss */  
[...]  
  
input.ng-invalid {  
    border-left-color: red;  
    border-left-style: solid;  
    border-left-width: 5px;  
}  
  
input.ng-valid {  
    border-left-color: green;  
    border-left-style: solid;  
    border-left-width: 5px;  
}
```

## Eigene Validierungsdirektiven

Eigene Validierungsregeln für Template-getriebene Formular stellt eine Anwendung über Direktiven bereit. Dabei handelt es sich um einen Building-Block, der mit der Komponente verwandt ist. Der einzige Unterschied besteht darin, dass Di-

rekiven kein Template haben. Sie werden also nicht direkt angezeigt, sondern können lediglich Verhalten einer Seite hinzufügen.

Die bereits in diesem Kapitel verwendete Direktive `ngModel` fügt zum Beispiel ein Datenbindungsverhalten zur Seite hinzu. Mit `*ngIf` blenden Sie Abschnitte ein und aus, und mit `ngClass` weisen Sie Klassen Elementen zu, um diese dynamisch mit Styles zu versehen.

Auch eine Validierungsdirektive fügt einer Seite Verhalten hinzu. Dieses besteht darin, der `ngModel`-Direktive eines Eingabefelds eine Validierungsregel bereitzustellen.

## Eine erste Validierungsdirektive erstellen

Zum Einstieg in die Welt der Validierungsdirektiven beschreiben wir hier eine einfache `CityValidationDirective`, die die bei der Flugsuche erfassten Städte validiert. Ein `appCity`-Attribut in den Eingabefeldern soll sie repräsentieren:

```
<input class="form-control"
       [(ngModel)]="from"
       name="from"
       required
       minlength="3"
       appCity>

<div *ngIf="f?.controls?.from?.hasError('appCity')">
  Dieser Flughafen existiert nicht
</div>
```

Das Grundgerüst dieser Direktive lässt sich mit der CLI generieren:

```
ng generate directive shared/validation/city-validation
```

Die Kurzform davon lautet:

```
ng g d shared/validation/city-validation
```

Wie alle anderen bisher mit der CLI generierten Building-Blocks lassen sich auch Direktiven direkt in Visual Studio Code mit dem Plug-in *Angular Schematics* generieren.

Um ein wenig Ordnung zu wahren, platzieren wir diese Direktive in einen Unterordner `validation` von `shared`. Somit registriert die CLI die Direktive auch beim Shared Module (siehe Beispiel 9-5).

*Beispiel 9-5: Die CLI deklariert und exportiert die CityValidationDirective im SharedModule.*

```
// src/app/shared/shared.module.ts
```

```
[...]
```

```
// Von der CLI eingefügt:
import { CityValidationDirective } from './validation/city-validation.directive';
```

```
@NgModule({
```

```

imports: [
  [...]
],
declarations: [
  [...]

  // Von der CLI eingefügt:
  CityValidationDirective
],
exports: [
  [...]

  // Neue Einträge:
  CityValidationDirective
]
})
export class SharedModule { }

```

Wie Komponenten und Pipes sind auch Direktiven unter declarations einzutragen. Diese Aufgabe übernimmt die CLI, allerdings ist es hier erneut eine gute Idee, sich dessen zu vergewissern. Damit wir die Direktive auch außerhalb des Moduls verwenden können, platzieren wir sie ebenfalls unter exports.



Sie können die CLI über den Schalter `--exports` anweisen, auch den nötigen Eintrag unter exports zu platzieren:

```
ng generate directive shared/validation/city-validation
--exports
```

Die gleiche Option bietet das *Angular Schematics*-Plug-in in Visual Studio Code.

Bei der generierten Direktive handelt es sich analog zu den anderen Building-Blocks von Angular um eine Klasse mit einem Directive-Dekorator (siehe Beispiel 9-6).

#### *Beispiel 9-6: Grundgerüst für einen eigenen Validator*

```
// src/app/shared/validation/city-validation.directive.ts

import { Directive } from '@angular/core';

@Directive({
  selector: '[appCityValidation]'
})
export class CityValidationDirective {

  constructor() { }

}
```

Der Directive-Dekorator offenbart das Verwandtschaftsverhältnis zur Komponente: Bis auf jene Eigenschaften, die sich auf das Template auswirken, bietet er die gleichen Einstellungsmöglichkeiten wie der Component-Dekorator an.

Unter Nutzung eckiger Klammern definiert der Selektor, dass diese Direktive das Verhalten sämtlicher Elemente mit einem Attribut `appCityValidation` erweitert.



Selektoren von Direktiven und Komponenten unterstützen die Möglichkeiten von CSS-Selektoren. Der Selektor

```
form input[appCityValidation]
```

würde beispielsweise sämtliche `input`-Elemente adressieren, die sich innerhalb eines `form`-Elements befinden und ein Attribut `appCityValidation` aufweisen.

In vereinzelten Fällen ist das auch nützlich. Allerdings ist es üblich, mit Selektoren von Komponenten auf Elementnamen und mit Selektoren von Direktiven auf Attributnamen bzw. auf eine Kombination aus Element- und Attributname zu verweisen.

Lassen Sie uns nun dieses Grundgerüst zu einer ersten Validierungsdirektive ausbauen. Dazu passen wir den Selektor ein wenig an und geben der Direktive einen Serviceprovider, mit dem sie sich selbst als Service für das Token `NG_VALIDATORS` registriert (siehe Beispiel 9-7).

*Beispiel 9-7: Eine einfache Validierungsdirektive*

```
// src/app/shared/validation/city-validation.directive.ts

// Importieren:
import { Directive } from '@angular/core';

// Diesen Import ergänzen:
import { Validator, AbstractControl, NG_VALIDATORS, ValidationErrors }
    from '@angular/forms';

@Directive({
    // Selektor aktualisieren:
    selector: 'input[appCity]',

    // Provider eintragen, um die Validierungsregel
    // beim aktuellen Eingabefeld zu registrieren:
    providers: [
        {
            provide: NG_VALIDATORS,
            useExisting: CityValidationDirective,
            multi: true
        }
    ]
})
export class CityValidationDirective implements Validator {

    // Das Interface Validator und somit
    // dessen Methode validate implementieren:
    public validate(c: AbstractControl): ValidationErrors | null {

        if (c.value === 'Graz'
```

```

    || c.value === 'Hamburg'
    || c.value === 'Frankfurt'
    || c.value === 'Wien'
    || c.value === 'Mallorca') {

        return null;
    }

    return {
        appCity: true
    };
}
}
// filename

```

Es handelt sich hierbei um einen Multi-Provider für das jeweilige Eingabefeld. Angular ruft zur Laufzeit sämtliche Services, die für `NG_VALIDATORS` registriert wurden, ab und nutzt deren `validate`-Methoden zum Validieren des jeweiligen Felds.

Damit Angular keine eigene Instanz der Validierungsdirektive für diesen Service-provider erstellt, sondern die aktuelle, die auch die Direktive repräsentiert, wieder verwendet, nutzt der Provider `useExisting` und nicht `useClass`.

Diese Klasse tritt also in zwei Rollen auf: Zum einen ist sie eine Direktive für ein Eingabefeld und zum anderen ein Service mit einer Validierungsfunktion.

Zugegeben, das ist ein wenig kompliziert. Manch einer hätte sich gegebenenfalls einen eigenen Building-Block für Validatoren erwartet. Auf der anderen Seite erlaubt diese Vorgehensweise, existierende Building-Blocks für weitere Anwendungsfälle zu nutzen.

Die Validierungsdirektive implementiert das Interface `Validator`, das die von Angular zur Laufzeit aufgerufene `validate`-Methode vorgibt. Diese nimmt ein `Abstract Control` entgegen, das das zu validierende Steuerelement repräsentiert. Der Rückgabewert ist ein `ValidationErrors`-Objekt. Dabei handelt es sich um ein Objekt, das als Dictionary verwendet wird. Jedes Schlüssel/Wert-Paar beschreibt einen Fehler, den Angular in der oben besprochenen Eigenschaft `errors` platziert.

Die betrachtete Implementierung prüft, ob es sich bei seinem Wert um einen bekannten Ort handelt. Wurde der Wert korrekt validiert, liefert sie `null`. Ansonsten gibt sie ein Fehlerbeschreibungsobjekt zurück, das mit dem Schlüssel `appCity` auf die fehlgeschlagene Validierung hinweist. Die Anwendung kann später prüfen, ob ein durch solch einen Schlüssel ausgedrückter Fehler vorliegt. Um die Übersicht zu wahren, bietet es sich an, diesen Schlüssel und das mit dem Selektor referenzierte HTML-Attribut gleich zu benennen.

Die Werte hinter diesem Schlüssel sind für Angular unerheblich, solange sie *truthy* sind. Allerdings kann die Anwendung sie nutzen, um weitere Informationen über den Fehler in Erfahrung zu bringen. Ist der Wert hingegen *falsy*, interpretiert Angular den Eintrag.



Als Alternative zur Nutzung eines Providers, der `NG_VALIDATORS` an eine Klasse mit einer `validate`-Methode bindet, könnte die Anwendung auch Gebrauch von einem Provider machen, der `NG_VALIDATORS` direkt an die Validierungsfunktion bindet:

```
@Directive({
  selector: 'input[appCity]',
  providers: [
    {
      provide: NG_VALIDATORS,
      useValue: validationFunction,
      multi: true,
    }
  ]
})
export class CityValidationDirective { }
```

Bei `validationFunction` handelt es sich um eine Funktion mit der Signatur der gezeigten Methode `validate`. Daneben können Sie mit `useValue` auch auf ein anderes Objekt, das solch eine `validate`-Methode anbietet, verweisen.

Um unsere Validierungsdirektive auszuprobieren, erweitern wir wie oben gezeigt das Eingabefeld für die Eigenschaft `from` (siehe Beispiel 9-8).

*Beispiel 9-8: Einsatz der Validierungsdirektive `appCity`*

```
<!-- src/app/flight-search/flight-search.component.html -->

[...]
<div class="form-group">
  <label>From:</label>
  <input
    [(ngModel)]="from"
    name="from"
    required
    minlength="3"
    appCity
    class="form-control">
</div>

[...]

<div class="error" *ngIf="f?.controls?.from?.hasError('appCity')">
  Die Stadt wird nicht angeflogen.
</div>

[...]
```

Erfassen Sie nun eine Stadt, die unser Validator nicht akzeptiert, erhalten Sie eine Fehlermeldung.

# Flight Search

From:

Gleisdorf

Es liegen Validierungsfehler für diese Eingabe vor.

Die Stadt wird nicht angeflogen.

Internes Errors-Objekt: { "appCity": true }

To:

Delayed

SEARCH

Abbildung 9-4: Die Validierungsregel appCity liefert einen Validierungsfehler.

Bitte beachten Sie auch, dass das errors-Objekt das von der Direktive gelieferte Schlüssel/Wert-Paar appCity: true aufweist. Um dem Formular mehr Informationen zur Validierung zukommen zu lassen, könnte die Direktive anstatt true auch ein Objekt als Wert festlegen:

```
// src/app/shared/validation/city-validation.directive.ts
[...]
return {
  appCity: {
    reason: 'Requested Airport is currently not available',
    allowedCities: ['Graz', 'Hamburg', [...]],
    tryAgain: 2031
  }
};
[...]
```

Diese Werte können Sie im Formular auslesen und z.B. im Rahmen der Validierungsfehler ausgeben:

```
<!-- src/app/flight-search/flight-search.component.html -->
[...]
{{ f?.controls?.from?.errors?.appCity?.allowedCities | json }}
[...]
```

## Parametrisierbare Validierungsdirektiven

Die Validierungsdirektive, die wir im letzten Abschnitt betrachtet haben, prüft gegen hartcodierte Werte. Die Lösung wäre flexibler, wenn sie gegen eine Liste frei definierbarer Einträge validieren würde:

```

<input
  [(ngModel)]="from"
  name="from"
  class="form-control"
  required
  appCity="Graz,München,Hamburg,Frankfurt,Zürich,Wien"
  minlength="3"
  maxlength="30">

```

Damit eine Direktive solche Parameter entgegennehmen kann, spendieren wir ihr dafür Eigenschaften. Analog zur Vorgehensweise bei Komponenten sind diese mit Input zu dekorieren (siehe Beispiel 9-9).

*Beispiel 9-9: Parametrisierbare Direktive*

```

// src/app/shared/validation/city-validation.directive.ts

// Importieren:
import { Directive, Input } from '@angular/core';

// Diesen Import ergänzen:
import { Validator, AbstractControl, NG_VALIDATORS, ValidationErrors }
    from '@angular/forms';

@Directive({
  // Selektor aktualisieren:
  selector: 'input[appCity]',

  // Provider eintragen:
  providers: [
    {
      provide: NG_VALIDATORS,
      useExisting: CityValidationDirective,
      multi: true
    }
  ]
})
export class CityValidationDirective implements Validator {

  @Input() appCity = '';

  // Das Interface Validator und somit
  // dessen Methode validate implementieren:
  public validate(c: AbstractControl): ValidationErrors | null {

    const allowedCities = this.appCity.split(',');

    if (allowedCities.includes(c.value)) {
      return null;
    }

    return {
      appCity: true
    };
  }
}

```

Es ist üblich, dass zumindest eine der Eigenschaften von Direktiven den gleichen Namen hat wie das Attribut, auf das sich der Selektor bezieht. Das erlaubt es, mit einem einzigen Attribut sowohl die Direktive zu aktivieren als auch einen Wert zuzuweisen:

```
<input  
  [(ngModel)]="from"  
  name="from"  
  class="form-control"  
  required  
  appCity="Graz,München,Hamburg,Frankfurt,Zürich,Wien"  
  minlength="3"  
  maxlength="30">
```

Neben dieser Eigenschaft lassen sich noch weitere einrichten:

```
[...]  
export class CityValidationDirective implements Validator {  
  
  @Input() appCity = '';  
  @Input() strict = false;  
  [...]  
}
```

Sämtliche Eigenschaften können beim Aufruf an Werte gebunden werden:

```
<input  
  [(ngModel)]="from"  
  name="from"  
  class="form-control"  
  required  
  appCity="Graz,München,Hamburg,Frankfurt,Zürich,Wien"  
  [strict]="true"  
  minlength="3"  
  maxlength="30">
```

Während appCity ohne eckige Klammern auskommt, ist das bei strict nicht der Fall. Das liegt daran, dass appCity lediglich einen hartcodierten String entgegennimmt. Möchten Sie hingegen einen JavaScript-Ausdruck übergeben oder eine Bindung mit einer Eigenschaft Ihrer Komponente erstellen, sind die eckigen Klammern notwendig.

Falls Sie auch für appCity eckige Klammern nutzen wollen, müssen Sie den übergebenen String mit Hochkommata begrenzen, da der Inhalt des Attributs in diesem Fall als JavaScript-Ausdruck ausgewertet wird:

```
<input  
  [...]  
  [appCity]="'Graz,München,Hamburg,Frankfurt,Zürich,Wien'"  
  [strict]="true"  
  [...]>
```

## Multi-Field-Validatoren erstellen

In einigen Fällen müssen Sie sicherstellen, dass die Werte verschiedener Felder miteinander korrelieren. Diese Aufgabe lässt sich lösen, indem Sie einen Validator auf der Ebene des gesamten Formulars bereitstellen. Dieser erhält Zugriff auf ein Form Group-Objekt, das strukturell der eingangs diskutierten NgForm-Direktive gleicht und Zugriff auf sämtliche FormControl-Objekte des Objektgraphen gewährt.

Um diese Idee zu demonstrieren, nutzen wir hier eine RoundTripValidationDirective, die Rundflüge verbietet. Dazu vergleicht sie die Felder from und to. Sie lässt sich mit der folgenden Anweisung oder mit dem Plug-in *Angular Schematics* einrichten:

```
ng generate directive shared/validation/round-trip-validation
```

Durch die Angabe des Ordners *shared* sollte die CLI die generierte Direktive im SharedModule unter declarations eintragen. Stellen Sie von Hand sicher, dass sie auch exportiert wird (siehe Beispiel 9-10).

*Beispiel 9-10: Das SharedModule deklariert die generierte RoundTripValidationDirective.*

```
// src/app/shared/shared.module.ts

[...]

// von der CLI eingefügt
import { RoundTripValidationDirective }
    from './validation/round-trip-validation.directive';

@NgModule({
  imports: [
    CommonModule,
    FormsModule,
  ],
  declarations: [
    [...]

    // von der CLI eingefügt
    RoundTripValidationDirective
  ],
  exports: [
    [...]
    // neue Einträge
    RoundTripValidationDirective
  ]
})
export class SharedModule { }
```

Die Implementierung dieser Validierungsdirektive findet sich in Beispiel 9-11.

*Beispiel 9-11: Der Multi-Field-Validator greift auf die einzelnen Felder des Formulars zu.*

```
// src/app/shared/validation/round-trip-validation.directive.ts

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

```

import { AbstractControl, FormGroup, NG_
VALIDATORS, ValidationErrors, Validator } from '@angular/forms';

@Directive({
  // angepasster Selektor
  selector: 'form[appRoundTrip]',
  providers: [
    {
      provide: NG_VALIDATORS,
      useExisting: RoundTripValidationDirective,
      multi: true
    }
  ]
})
export class RoundTripValidationDirective implements Validator {

  constructor() { }

  validate(control: AbstractControl): ValidationErrors | null {
    const group = control as FormGroup;

    const from = group.controls?.from?.value;
    const to = group.controls?.to?.value;

    // Erst validieren, wenn Werte für beide Felder vorliegen:
    if (!from || !to) {
      return null;
    }

    if (from !== to) {
      // kein Round Trip: alles okay
      return null;
    }

    // Round Trip: Fehler melden
    return {
      appRoundTrip: true
    };
  }
}

```

Die vom Interface Validator vorgegebene validate-Methode nimmt per definitio-nem ein AbstractControl entgegen. Wenden wir den Validator auf ein gesamtes Formular an, erhalten wir hier jedoch eine Instanz von FormGroup. Dabei handelt es sich um eine Subklasse von AbstractControl. Um Zugriff auf dessen zusätzliche Ei-genschaften zu erhalten, ist eine Type Assertion (as) notwendig.

Wie bereits die NgForm-Direktive in den vorangegangenen Beispielen gibt uns die Form Group über eine controls-Auflistung Zugriff auf die einzelnen FormControl-Instanzen.

Um diese Validierungsdirektive zu nutzen, ist sie direkt auf das `form`-Element unseres Suchformulars anzuwenden (siehe Beispiel 9-12).

*Beispiel 9-12: Der Multi-Field-Validator wird auf das form-Element angewendet.*

```
<!-- src/app/flight-search/flight-search.component.html -->

<h1>Flight Search</h1>

<form #f="ngForm" appRoundTrip>

  <div class="error" *ngIf="f?.errors">
    {{ f?.errors | json }}
  </div>

  <div class="error" *ngIf="f?.hasError('appRoundTrip')">
    Round Trips are not supported!
  </div>

  [...]
</form>
[...]
```

Die `NgForm`-Direktive bietet auch ein `errors`-Objekt sowie eine Methode `hasError` für Fehler auf Formularebene. Damit lässt sich herausfinden, ob unsere `RoundTrip ValidationDirective` einen Fehler gemeldet hat.

Das Ergebnis gestaltet sich, wie in Abbildung 9-5 gezeigt.

The screenshot shows a web page titled "Flight Search". Below the title, there is an error message: `{"appRoundTrip": true}` and "Round-Trips sind leider nicht möglich!". There are two input fields labeled "From:" and "To:", both containing the text "Hamburg". Below these fields is a checkbox labeled "Delayed" which is unchecked. At the bottom is a large "SEARCH" button.

Abbildung 9-5: Fehlermeldung des Multi-Field-Validators

## Asynchrone Validatoren

In einigen Fällen kann ein Validator nicht sofort feststellen, ob der Wert des übergebenen Felds korrekt ist. Dies ist zum Beispiel dann der Fall, wenn er dazu eine

serverseitige Routine anstoßen muss. Für solche Fälle sieht Angular asynchrone Validatoren vor. Diese antworten zunächst nur mit einem Promise oder einem Observable, das das Fehlerbeschreibungsobjekt später nachreicht.

Als Beispiel erstelle ich hier eine Direktive, die unseren FlightService verwendet, um zu prüfen, ob für die erfasste Stadt überhaupt Flüge vorliegen.

Zum Generieren der Direktive nutzen wir wieder die CLI:

```
ng generate directive shared/validation/async-city-validation
```

Stellen Sie bitte sicher, dass die neue Direktive im SharedModule sowohl unter declarations als auch unter exports eingetragen ist.

Die Implementierung dieser Direktive findet sich in Beispiel 9-13.

*Beispiel 9-13: Direktive mit asynchroner Validierungsregel*

```
// src/app/shared/validation/async-city-validation.directive.ts

import { Directive } from '@angular/core';
import { AbstractControl, AsyncValidator, NG_ASYNC_VALIDATORS, ValidationErrors } from '@angular/forms';
import { Observable } from 'rxjs';

// Eventuell müssen Sie diesen Import manuell einfügen:
import { delay, map } from 'rxjs/operators';

import { FlightService } from 'src/app/flight-booking/flight.service';

@Directive({
  // angepasster Selektor
  selector: '[appAsyncCity]',

  // Provider für NG_ASYNC_VALIDATORS:
  providers: [
    {
      provide: NG_ASYNC_VALIDATORS,
      useExisting: AsyncCityValidationDirective,
      multi: true
    }
  ]
})
export class AsyncCityValidationDirective implements AsyncValidator {

  // FlightService injizieren:
  constructor(private flightService: FlightService) { }

  // Implementierung des Interface AsyncValidator
  validate(control: AbstractControl): Promise<ValidationErrors | null> | Observable<ValidationErrors | null> {

    const error: ValidationErrors = { appAsyncCity: true };

    return this.flightService.find(control.value, '').pipe(
```

```

        map(flights => flights.length === 0 ? error : null),
        delay(2000)
    );
}

}

```

Im Wesentlichen weist eine asynchrone Validierungsdirektive zwei Unterschiede gegenüber ihren synchronen Gegenstücken auf:

- Sie implementiert das Interface `AsyncValidator`, dessen `validate`-Methode einen Promise oder ein Observable zurückliefert. Diese Datenstrukturen veröffentlichen das `ValidationErrors`-Objekt, sobald sie vorliegen.
- Sie registrieren sich selbst unter dem von Angular vorgegebenen Token `NG_ASYNC_VALIDATORS`.

Das betrachtete Beispiel lässt sich den `FlightService` injizieren und prüft damit, ob es Flüge für den zu validierenden Flughafen gibt. Dazu ruft es sämtliche Flüge für diesen Flughafen ab. Zugegeben, diese Implementierung ist ein wenig grobgranal, zumal wir die abgerufenen Flüge gar nicht benötigen, sondern nur wissen müssen, ob es Flüge gibt. Zur Vereinfachung dieses Beispiels greifen wir hier trotzdem auf die bestehende `find`-Methode zurück.

Das Ergebnis von `find` ist ein Observable mit den gefundenen Flügen. Die Methode `map` transformiert es in ein Observable mit einem `ValidationErrors`-Objekt, wenn das Array leer ist. Ansonsten transformiert `map` es in ein Observable mit dem Wert `null`. Um die Asynchronität dieses Vorgangs zu veranschaulichen, führt `delay` danach zu einer künstlichen Verzögerung von zwei Sekunden. Daraus ergibt sich ein weiteres Observable, das `validate` zurückliefert.



Mehr Details zur Arbeit mit Observables finden Sie in Kapitel 11.

Um die Direktive zu nutzen, ist das Eingabefeld für die Eigenschaft `from` um das definierte `appAsyncCity`-Attribut zu ergänzen:

```

<!-- src/app/flight-search/flight-search.component.html -->

[...]

<input [(ngModel)]="from"
       name="from"
       required
       minlength="3"
       appAsyncCity
       appCity="Tripsdrill,Graz,München,Hamburg,Frankfurt,Zürich,Wien"
       class="form-control">

[...]

```

Beim Einsatz asynchroner Validierungsregeln ist es wichtig, zu wissen, dass Angular diese nur anstößt, wenn kein synchroner Validator einen Fehler gemeldet hat. Aus diesem Grund haben wir hier den Ort *Tripsdrill* für den synchronen appCity-Validator konfiguriert. Dieser Ort wird somit vom synchronen Validator akzeptiert und vom asynchronen abgewiesen, sofern niemand in unserer öffentlichen Demo-API einen Flug für diesen Ort einträgt.

Wenn Sie sich damit nicht belasten wollen, können Sie das Attribut appCity auch entfernen. Danach können Sie mit hasError prüfen, ob der Fehler appAsyncCity vorliegt:

```
<div class="error" *ngIf="f?.controls?.from?.hasError('appAsyncCity')">  
  Die Stadt wird nicht angeflogen.  
</div>
```

Während noch mindestens ein asynchroner Validator läuft, setzt Angular eine Eigenschaft pending auf true:

```
<div class="error" *ngIf="f?.controls?.from?.pending">  
  Validierung wird ausgeführt!  
</div>
```

Die Eigenschaft valid, die wir bis jetzt als Boolean angesehen haben, hat bei Nutzung asynchroner Validierungsregel drei mögliche Werte:

- true: Das Formular bzw. das Feld ist valide.
- false: Das Formular bzw. das Feld ist nicht valide.
- undefined: Es werden noch asynchrone Validierungsregeln ausgeführt. Deswegen wissen wir noch nicht, ob das Formular bzw. das Feld valide ist.

Aus diesem Grund kann es notwendig sein, explizit auf einen dieser Werte zu prüfen:

```
<!-- Bisherige Nutzung -->  
<!-- <div class="error" *ngIf="!f?.controls?.from?.valid"> -->  
  
<!-- Aktuelle Nutzung -->  
<div class="error" *ngIf="f?.controls?.from?.valid === false">  
  Es liegen Validierungsfehler für diese Eingabe vor.  
</div>
```

Nachdem Sie diesen Validator ausprobiert haben, empfiehlt es sich, das in der Datei *async-city-validation.directive.ts* eingeführte delay zu entfernen.

## Komponente zum Präsentieren von Validierungsfehlern

Um nicht pro Eingabefeld immer und immer wieder auf dieselbe Weise die Validierungsfehler abfragen zu müssen, bietet sich der Einsatz einer zentralen Komponente an. Diese kann die Eigenschaft errors des zu validierenden FormControl entgegennehmen und basierend auf seinen Schlüssel/Wert-Paaren Fehlermeldungen präsentieren.

Diese Idee möchten wir hier mit einer `ValidationErrorsComponent` veranschaulichen. Diese haben wir mit der CLI für das `SharedModule` generiert und exportiert:

```
ng generate component shared/validation/validation-errors --export
```

Natürlich können Sie auch hier das Visual-Studio-Code-Plug-in *Angular Schematics* verwenden. Bitte stellen Sie sicher, dass die neue Komponente im `SharedModule` sowohl unter `declarations` als auch unter `exports` zu finden ist.

Die Implementierung der Komponente, die lediglich das `errors`-Objekt des validierten Felds entgegennimmt, findet sich in Beispiel 9-14.

*Beispiel 9-14: Komponente zum Anzeigen von Validierungsfehlern*

```
// src/app/shared/validation/validation-errors/validation-errors.component.ts

import { Component, Input, OnInit } from '@angular/core';
import { ValidationErrors } from '@angular/forms';

@Component({
  selector: 'app-validation-errors',
  templateUrl: './validation-errors.component.html',
  styleUrls: ['./validation-errors.component.scss']
})
export class ValidationErrorsComponent implements OnInit {

  @Input() errors: ValidationErrors | undefined | null;

  constructor() { }

  ngOnInit(): void {
  }

}
```

Da nicht zu jedem Zeitpunkt für jedes Feld ein `errors`-Objekt vorliegt, akzeptiert `errors` auch `undefined`. Da das Vorliegen keiner Fehler durch `null` repräsentiert wird, akzeptiert es auch diesen Wert. Das bedeutet, dass wir im Template mit dem Safe-Access-Operator arbeiten müssen (siehe Beispiel 9-15).

*Beispiel 9-15: Das Template der `ValidationErrorsComponent` präsentiert die entdeckten Validierungsfehler.*

```
<!-- src/app/shared/validation/validation-errors/validation-errors.component.html -->

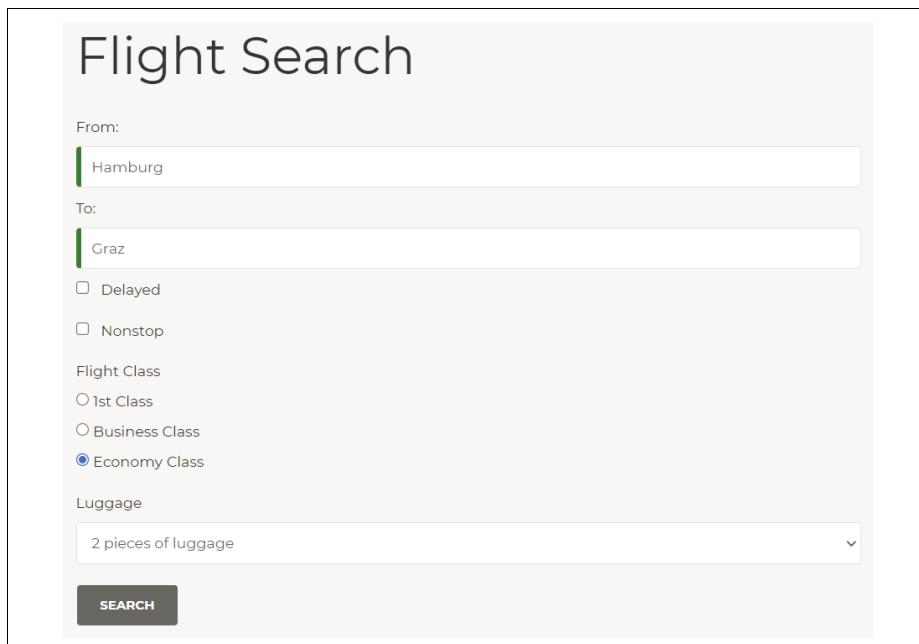
<div *ngIf="errors?.required" class="error">
  This field is required
</div>
<div *ngIf="errors?.minlength" class="error">
  Please enter at least {{ errors?.minlength?.requiredLength }} characters
</div>
<div *ngIf="errors?.appCity" class="error">
  This city is now allowed
</div>
```

Aus Gründen der Übersicht beschränken wir uns hier auf drei Prüfungen. Nun müssen wir lediglich diese Komponente pro Eingabefeld aufrufen:

```
<!-- src/app/flight-search/flight-search.component.html -->  
[...]  
      name="from"  
      required  
      minlength="3"  
      appAsyncCity  
      appCity="Tripsdrill,Graz,München,Hamburg,..."  
      class="form-control"/>  
  
<app-validation-errors [errors]="f?.controls?.from?.errors">  
</app-validation-errors>  
[...]
```

## Die Standardsteuerelemente von HTML nutzen

Bis jetzt haben die Beispiele lediglich Formulare mit Textboxen verwendet. Die Direktive `ngModel` kann sich allerdings ab Werk auch an alle anderen durch HTML definierten Standardsteuerelemente binden. Um diese Möglichkeiten zu demonstrieren, kommt hier eine erweiterte Version des verwendeten Suchformulars zum Einsatz (siehe Abbildung 9-6):



The screenshot shows a web-based flight search interface. At the top, it says "Flight Search". Below that, there are two input fields: "From:" containing "Hamburg" and "To:" containing "Graz". Underneath these are two checkboxes: "Delayed" and "Nonstop", both of which are unchecked. There are three radio buttons labeled "Flight Class": "1st Class", "Business Class", and "Economy Class", with "Economy Class" being the selected option (indicated by a blue dot). Below these is a section for "Luggage" with a dropdown menu showing "2 pieces of luggage". At the bottom of the form is a large, dark grey "SEARCH" button.

Abbildung 9-6: Erweiterte Flugsuche



Auch eigene Komponenten lassen sich zum Zusammenspiel mit ngModel bewegen. In Kapitel 18 gehen wir darauf ein.

## Checkboxen

Der Einsatz von Checkboxen gestaltet sich sehr geradlinig: Sie lassen sich mit ngModel direkt an Booleans binden. Um das zu demonstrieren, erweitern wir die FlightSearchComponent um die boolesche Eigenschaft nonstop:

```
nonstop: boolean;
```

Anschließend kann diese Eigenschaft an ein input-Feld mit dem Typ checkbox gebunden werden:

```
<div class="form-group">
  <label>
    <input [(ngModel)]="nonstop" type="checkbox" name="nonstop">
    &nbsp;
    Nonstop
  </label>
</div>
```

## Radiobuttons

Um den Einsatz von Radiobuttons zu demonstrieren, kommt hier das Interface FlightClass zum Einsatz:

```
// src/app/flight-booking/flight-class.ts

export interface FlightClass {
  id: number;
  name: string;
}
```

Daneben erhält die FlightSearchComponent ein Array dieses Interface, das die mögliche Auswahl an Flugklassen repräsentiert:

```
// src/app/flight-search/flight-search.component.ts
[...]

import { FlightClass } from '../flight-class';

[...]
@Component([...])
export class FlightSearchComponent implements OnInit {
  [...]

  // Auswahl
  flightClasses: FlightClass[] = [
    { id: 1, name: '1st Class' },
    { id: 2, name: 'Business Class' },
```

```

    { id: 3, name: 'Economy Class' }
];
// tatsächlich ausgewählte Klasse mit Standardwert
flightClass = this.flightClasses[2];
[...]
}

```

Die tatsächlich ausgewählte Option verstaut die Komponente in der Eigenschaft `flightClass`.

Um nun einen Radiobutton für sämtliche Flugklassen zu präsentieren, iteriert das Template über das Array mit den Vorschlagswerten:

```

<div class="form-group">
  <label>Flight Class</label>
  <div *ngFor="let c of flightClasses">
    <label>
      <input
        type="radio"
        name="flightClass"
        [(ngModel)]="flightClass"
        [value]="c">
      {{c.name}}
    </label>
  </div>
</div>

```

Pro Array-Eintrag erzeugt es ein `input`-Feld mit dem Typ `radio`. An dessen Eigenschaft `value` bindet es den jeweiligen Vorschlagswert, und mit `ngModel` bindet es das Feld `flightClass`, das den ausgewählten Wert widerspiegelt.

## Drop-down-Felder

Die Vorgehensweise beim Binden von Drop-down-Feldern ist zunächst ähnlich der bei Radiobuttons. Für die einzelnen Optionen erhält die Anwendung zunächst ein Interface `LuggageOption`:

```

// src/app/flight-booking/luggage-option.ts

export interface LuggageOption {
  id: number;
  name: string;
}

```

Ein Array von diesem Typ spiegelt in der Komponente die möglichen Vorschlagswerte wider:

```

// src/app/flight-search/flight-search.component.ts
[...]
import { FlightClass } from '../flight-class';

```

```

[...]
@Component([...])
export class FlightSearchComponent implements OnInit {
  [...]

  // Auswahl
  luggageOptions: LuggageOption[] = [
    { id: 0, name: 'No luggage' },
    { id: 1, name: '1 piece of luggage' },
    { id: 2, name: '2 pieces of luggage' }
  ];

  // tatsächlich ausgewählte Option mit Standardwert
  luggage = this.luggageOptions[2];

  [...]
}

```

Die Eigenschaft `luggage` nimmt den ausgewählten Wert auf.

Um das Drop-down-Feld aufzubauen, erzeugt das Template ein `select`-Element. Darin platziert es pro Option ein `option`-Element:

```

<div class="form-group">
  <label>Luggage</label>
  <select [(ngModel)]="luggage" name="luggage" class="form-control">
    <option *ngFor="let l of luggageOptions" [ngValue]="l">{ l.name }</option>
  </select>
</div>

```

Die Direktive `ngValue` bindet den zuzuweisenden Wert an dieses Element. Der Einsatz dieser Direktive ist notwendig, da das DOM-Attribut `value` lediglich `string` aufnehmen kann, hier jedoch eine `LuggageOption` zum Einsatz kommt. Daneben bindet `ngModel` die Eigenschaft `luggage`, die das ausgewählte Objekt erhalten soll, an das `select`-Element.

## Zusammenfassung

Angular nutzt einen Objektgraphen, der die verwalteten Formulare beschreibt. Die darin zu findenden Objekte geben Auskunft über die aktuellen Werte der Felder oder das Ergebnis der Validierung. Nutzt eine Anwendung Template-getriebene Formulare, generiert Angular diesen Objektgraphen aus dem Template. Bei reaktiven Formularen ist es hingegen die Aufgabe des Entwicklers, die Objekte bereitzustellen.

Bei diesem Objektgraphen lassen sich auch Validierungsregeln hinterlegen. Dazu ist bei Template-getriebenen Formularen lediglich ein Attribut im Markup des zu validierenden Felds zu hinterlegen. Eigene Validierungsregeln sind Direktiven, die für ein Feld einen Service mit einer `validate`-Methode registrieren. Angular ruft zum Validieren alle diese Services ab und führt deren `validate`-Methoden aus.

Synchrone Validatoren liefern sofort das Ergebnis der Validierung. Asynchrone Validatoren liefern das Ergebnis nach, sobald es vorliegt. Diese Art von Validatoren ist notwendig, wenn man sich serverseitig erkundigen muss, ob der gegebene Wert korrekt ist.

# Reaktive Formulare

Nachdem wir im letzten Kapitel die Template-getriebenen Formulare näher betrachtet haben, wenden wir uns hier ihren reaktiven Gegenstücken zu. Das Angular-Team nennt diese Formulare *reakтив*, weil sie mithilfe von Observables über Zustandsänderungen informieren. Die Anwendung kann also augenblicklich darauf reagieren.

Die wohl wichtigere Eigenschaft dieser Spielart ist jedoch, dass die Anwendung selbst den Objektgraphen für das Formular und dessen Felder aufbauen muss. Das ist zunächst ein Mehraufwand, hilft aber, den Überblick zu bewahren. Bei einem Blick auf den Code des letzten Kapitels erkennt man zum Beispiel, dass die einfacheren Template-getriebenen Formulare das Template aufzublühen. Mit reaktiven Formularen verlagern Sie hingegen die meisten dieser Aspekte in die Komponentenklasse oder die Services. Dort lässt er sich besser strukturieren und automatisiert testen.

In diesem Kapitel gehen wir auf diese Unterschiede ein und zeigen, wie Sie reaktive Formulare nutzen können.

## Erste Schritte mit reaktiven Formularen

Im Folgenden lernen Sie zunächst, mit reaktiven Formularen umzugehen. Dazu erweitern wir die bestehende `FlightEditComponent`.

### Modul einbinden

Um reaktive Formulare nutzen zu können, müssen Sie das Angular-Modul `ReactiveFormsModule` in die gewünschten Angular-Module aufnehmen (siehe Beispiel 10-1).

*Beispiel 10-1: Einbinden des ReactiveFormsModule*

```
// src/app/flight-booking/flight-booking.module.ts
```

```
[...]
```

```

// ReactiveFormsModule hinzufügen:
import { FormsModule, ReactiveFormsModule } from '@angular/forms';

@NgModule({
  imports: [
    [...]
    // Einfügen:
    ReactiveFormsModule
  ],
  declarations: [
    [...]
  ],
  exports: [
    FlightSearchComponent
  ]
})
export class FlightBookingModule { }

```

## Das Formular mit einem Objektgraphen beschreiben

Wenn Sie reaktive Formulare nutzen, ist die Komponente für das Erstellen des Objektgraphen zum Beschreiben des Formulars verantwortlich. Tabelle 10-1 zeigt die Klassen, die Ihnen dafür zur Verfügung stehen.

*Tabelle 10-1: Klassen zur Beschreibung von Formularen*

Klasse	Beschreibung
AbstractControl	Abstrakte Klasse, von der die nachfolgenden Klassen ableiten.
FormControl	Beschreibt ein einzelnes Eingabefeld.
FormGroup	Gruppierung von AbstractControl. Beschreibt ein gesamtes Formular oder zumindest einen Abschnitt eines Formulars mit Schlüssel/Wert-Paaren. Die Schlüssel sind interne Feldbezeichner, und die Werte sind AbstractControls.
FormArray	Array mit AbstractControls. Repräsentiert Wiederholgruppen.

Einige dieser Klassen dürften Ihnen schon aus dem Kapitel zu Template-getriebenen Formularen bekannt sein. Der für Sie generierte Objektgraph stützt sich nämlich auch darauf.

Um ein Formular für einen Flug zu beschreiben, könnte der Controller eine Form Group instanziieren und ihr entsprechende FormControls zuweisen. Beispiel 10-2 veranschaulicht das im Rahmen unserer FlightEditComponent.

*Beispiel 10-2: Objektgraph für ein reaktives Formular erstellen*

```
// src/app/flight-booking/flight-edit/flight-edit.component.ts
```

```
[...]
```

```
// Hinzufügen:
```

```
import { FormControl, FormGroup, Validators } from '@angular/forms';
```

```

@Component([...])
export class FlightEditComponent implements OnInit {

    [...]

    formGroup: FormGroup;

    constructor(private route: ActivatedRoute) {

        // Hinzufügen:
        this.formGroup = new FormGroup({
            id: new FormControl(),
            from: new FormControl('Graz',
                [Validators.required, Validators.minLength(3)]),
            to: new FormControl('Hamburg'),
            date: new FormControl(),
            delayed: new FormControl(false)
        });

        this.formGroup.statusChanges.subscribe(
            value => console.debug('whole form changed', value)
        );

        this.formGroup.controls.delayed.statusChanges.subscribe(
            value => console.debug('delayed changed', value)
        );
    }

    save(): void {
        console.debug('form to save', this.formGroup.value);
        console.debug('id', this.formGroup.controls.id.value);
    }
}

```

Das Beispiel übergibt an den Konstruktor von `FormGroup` ein Objekt, das Feldnamen auf deren `FormControl`s abbildet. Die Felder `from` und `to` bekommen einen Standardwert. Das Feld `from` bekommt zusätzlich zwei Validatoren zugewiesen. Dabei handelt es sich um statische Methoden in der von Angular bereitgestellten `Validators`-Klasse.

Gleich im Anschluss daran nutzt das Beispiel den aufgebauten Objektgraphen und registriert sich mit `subscribe` für Änderungen am gesamten Formular sowie an der Eigenschaft `delayed`.

Immer wenn sich das gesamte Formular ändert, veröffentlicht Angular ein Objekt, das die Namen der Formularfelder auf deren aktuelle Werte abbildet:

```
{
    id: 1,
    from: 'Graz',
```

```

        to: 'Hamburg',
        date: '...',
        delayed: false
    }

```

Mit ein wenig Glück ist das schon jene Repräsentation, die der Server beim Erzeugen und Aktualisieren verlangt.

Die darunter gezeigte Methode `save` gibt auch die Werte des gesamten Formulars sowie darunter den Wert des Felds `id` aus.

## Reactive Formulare mit dem FormBuilder beschreiben

Damit wir nicht wie im letzten Abschnitt pro Eingabefeld manuell ein `FormControl` instanziieren müssen, bietet Angular mit dem `FormBuilder` eine Vereinfachung. Es handelt sich dabei um einen Service mit Hilfsmethoden. Beispiel 10-3 zeigt, wie sich damit ein Formular zum Verwalten eines Flugs beschreiben lässt.

*Beispiel 10-3: Beschreibung eines Formulars mit dem FormBuilder*

```
// src/app/flight-booking/flight-edit/flight-edit.component.ts

[...]

// Auch FormBuilder importieren:
import { FormBuilder, FormControl, FormGroup, Validators } from '@angular/forms';

@Component([...])
export class FlightEditComponent implements OnInit {

    [...]

    formGroup: FormGroup;

    // FormBuilder injizieren lassen:
    constructor(bf {
        private route: ActivatedRoute,
        private fb: FormBuilder) {

        // Mit FormBuilder Objektgraphen erzeugen:
        this.formGroup = fb.group({
            id: [],
            from: [
                'Graz',
                [Validators.required, Validators.minLength(3)]
            ],
            to: ['Hamburg'],
            date: [],
            delayed: []
        });
    }

    this.formGroup.controls.delayed.statusChanges.subscribe(
        value => console.debug('delayed changed', value)
    )
}
```

```

    );
    this.formGroup.statusChanges.subscribe(
      value => console.debug('whole form changed', value)
    );
  }

  save(): void {
    console.debug('form to save', this.formGroup.value);
    console.debug('id', this.formGroup.controls.id.value);
  }
}

```

Das Beispiel fordert per Dependency Injection einen `FormBuilder` an und erstellt mit dessen Methode `group` eine `FormGroup`, die das gesamte Formular widerspiegelt.

Die Namen der Eigenschaften des an `group` übergebenen Objekts stehen für die Namen der Steuerelemente, die Sie einrichten wollen. Das dahinterstehende Array beschreibt das jeweilige Feld näher. Per definitionem stellt der erste Array-Eintrag den Standardwert des Felds dar.

Der zweite Eintrag kann einen Validator beinhalten, der den Feldinhalt prüft. Möchte der Entwickler mehrere Validatoren nutzen, muss er diese – in einem Array verpackt – hier angeben.

Sämtliche Array-Einträge sind optional: Es müssen weder Standardwerte noch Validatoren hinterlegt werden.

## Einen Objektgraphen an ein Formular binden

Da bereits die Klasse der Komponente die meisten Informationen für reaktive Formulare definiert, gestaltet sich das Markup entsprechend schlanker. Im hier betrachteten Fallbeispiel muss das Template lediglich ein Formular mit der von uns bereitgestellten `FormGroup` verknüpfen (siehe Beispiel 10-4).

*Beispiel 10-4: Einen Objektgraphen an ein Formular binden*

```

<!-- src/app/flight-booking/flight-edit/flight-edit.component.html -->

[...]

<form [formGroup]="formGroup">

  <div class="form-group">
    <label>Id:</label>
    <input formControlName="id" class="form-control">
  </div>

  <div class="form-group">
    <label>From:</label>

```

```

<input formControlName="from" class="form-control">
<div class="error" *ngIf="!formGroup.controls.from.valid">
    Validierungsfehler!
</div>
<div class="error" *ngIf="formGroup.controls.from.hasError('required')">
    Pflichtfeld!
</div>
<div class="error" *ngIf="formGroup.controls.from.hasError('minlength')">
    Min. 3 Zeichen!
</div>
</div>

<div class="form-group">
    <label>To:</label>
    <input formControlName="to" class="form-control">
</div>

<div class="form-group">
    <label>Date:</label>
    <input formControlName="date" class="form-control">
</div>

<div class="form-group">
    <input formControlName="delayed" type="checkbox">
    &nbsp;
    <label>Delayed:</label>
</div>

<button class="btn btn-default" (click)="save()">Save</button>

</form>

```

Die Verknüpfung mit der `FormGroup` erfolgt mit der `formGroup`-Direktive. Die Verknüpfung mit den Eingabefeldern geschieht hingegen mit der Direktive `formControlName`, die lediglich den Namen des gewünschten `FormControl`-Objekts erhält. Um das gewünschte Objekt zu finden, durchforstet sie mit diesem Namen die umgebende `FormGroup`.

Das gezeigte Template verwendet die Eigenschaften der `FormGroup` sowie der `FormControls`, um sich über Validierungsfehler zu informieren. Der dazu notwendige Zugriffspfad entspricht jenem, den wir schon von Template-getriebenen Formularen kennen. Da der Objektgraph bei reaktiven Formularen jedoch von Anfang an zur Verfügung steht, ist der Einsatz des Safe-Access-Operators nicht notwendig.

Zum Ausprobieren der Lösung starten wir `ng serve` und navigieren dann zur `Flight EditComponent` (siehe Abbildung 10-1).

Um den Weg abzukürzen, können Sie auch direkt die URL der zugrunde liegenden Route verwenden:

```
http://localhost:4200/flight-booking/flight-edit/3;showDetails=false
```

The screenshot shows a web application window titled 'FlightApp' with the URL 'http://localhost:4200/flight-booking/flight-edit/3;showDetails=false'. The main title is 'FLIGHTS 42' and the sub-section is 'FLIGHT BOOKING'. The page is titled 'Flight Edit' and displays a form with the following fields:

- Id: 3
- ShowDetail: false
- Id: (empty input field)
- From: Graz
- To: Hamburg
- Date: (empty input field)
- Delayed: (checkbox is unchecked)

A 'SAVE' button is at the bottom.

Abbildung 10-1: Reaktives Formular

## Werte ins Formular schreiben

Um das Formular mit Werten zu bestücken, können Sie die Methode `patchValue` der `FormGroup` aufrufen:

```
this.formGroup.patchValue({  
  id: 17,  
  from: 'Hier',  
  to: 'Da',  
  date: new Date().toISOString(),  
  delayed: false  
});
```

Diese Methode nimmt ein Objekt entgegen, das zur Struktur des Formulars passen muss. Vor allem müssen die verwendeten Eigenschaftsnamen den Namen der Form Controls entsprechen.

Das übergebene Objekt muss nicht für alle Felder einen Wert aufweisen. Angular unterstützt auch Objekte, die lediglich für eine Untergruppe der Felder Werte beinhalten.

Alternativ dazu können Sie die einzelnen Werte einzeln setzen:

```
this.formGroup.controls.id.value = 1701;
```

## Validatoren

Die letzten Abschnitte haben bereits die vordefinierten Validatoren `required` und `minlength` genutzt. In diesem Abschnitt zeigen wir, wie Sie eigene Validatoren für reaktive Formulare entwickeln können.

## Synchrone Validatoren

Bei Validatoren für reaktive Formulare handelt es sich lediglich um statische Methoden oder Funktionen, die ein Control entgegennehmen und ein Fehlerbeschreibungsobjekt zurückliefern. Wie von den Template-getriebenen Formularen bekannt, beinhaltet dieses Objekt pro Fehler ein Schlüssel/Wert-Paar. Konnte der Validator keinen Fehler entdecken, liefert er einfach ein leeres Objekt oder null.

Beispiel 10-5 zeigt das Beispiel eines benutzerdefinierten Validators für ein reaktives Formular. Anders als jene, die Angular über die Klasse Validators bereitstellt, handelt es sich dabei nicht um eine statische Methode, sondern um eine Funktion.

*Beispiel 10-5: Eigener Validator für reaktive Formulare*

```
// src/app/shared/validation/reactive/city-validator.ts

import { ValidatorFn } from '@angular/forms';

export const cityValidator: ValidatorFn = (control) => {

  if (control.value === 'Graz'
    || control.value === 'Hamburg'
    || control.value === 'Frankfurt'
    || control.value === 'Berlin'
    || control.value === 'Wien') {

    return null;
  }

  return {
    city: true
  };
};
```

Der Typ ValidatorFn erzwingt die Signatur, die die reaktiven Formulare unterstützen. Das übergebene Control-Objekt ist vom Typ AbstractControl. Aufgrund der Typerleitung von TypeScript muss das im Code jedoch nicht explizit erwähnt werden.

Um diesen Validator zu verwenden, müssen Sie ihn beim jeweiligen FormControl hinterlegen (siehe Beispiel 10-6).

*Beispiel 10-6: Validator für reaktives Formular nutzen*

```
// src/app/flight-booking/flight-edit/flight-edit.component.ts

[...]

// Import hinzufügen:
import { cityValidator } from '../../../../../shared/validation/reactive/city-validator';

[...]
```

```

this.formGroup = fb.group({
  id: [],
  from: [
    'Graz',
    [
      Validators.required,
      Validators.minLength(3),
      // Validator registrieren
      cityValidator
    ]
  ],
  to: ['Hamburg'],
  date: [],
  delayed: []
});
[...]

```

Bitte beachten Sie, dass `cityValidator` ohne runde Klammern verwendet wird. Das Beispiel ruft den Validator somit nicht auf, sondern hinterlegt eine Referenz auf ihn im Objektgraphen. Somit kann Angular bei Bedarf den Validator anstoßen.

Ob der Validator einen Fehler gefunden hat, lässt sich – wie gewohnt – der `valid`-Eigenschaft des jeweiligen Controls entnehmen:

```

<!-- src/app/flight-booking/flight-edit/flight-edit.component.html -->
[...]
<div class="error" *ngIf="formGroup.controls.from.hasError('city')">
  This city is not supported!
</div>
[...]

```

Bei `city` handelt es sich um den Schlüssel des vom Validator zurückgelieferten Schlüssel/Wert-Paars.

Wenn Sie die Anwendung ausführen und zur `FlightEditComponent` wechseln, sollte sich das in Abbildung 10-2 gezeigte Verhalten ergeben.



Standardmäßig findet die Validierung bei jedem einzelnen Tastaturanschlag statt. Dieses Verhalten lässt sich mit dem zweiten optionalen Parameter der Methode `group` anpassen:

```

this.formGroup = fb.group({
  id: [],
  [...]
}, {
  updateOn: 'blur',
});

```

Der Wert `blur` legt fest, dass die Validierung erst beim Verlassen der Felder erfolgen soll. Mit `submit` lässt sich Angular hingegen anweisen, erst beim Betätigen einer Schaltfläche im Formular zu validieren. Der Standardwert, der die Validierung bei jedem Anschlag bewirkt, lautet auf `change`.

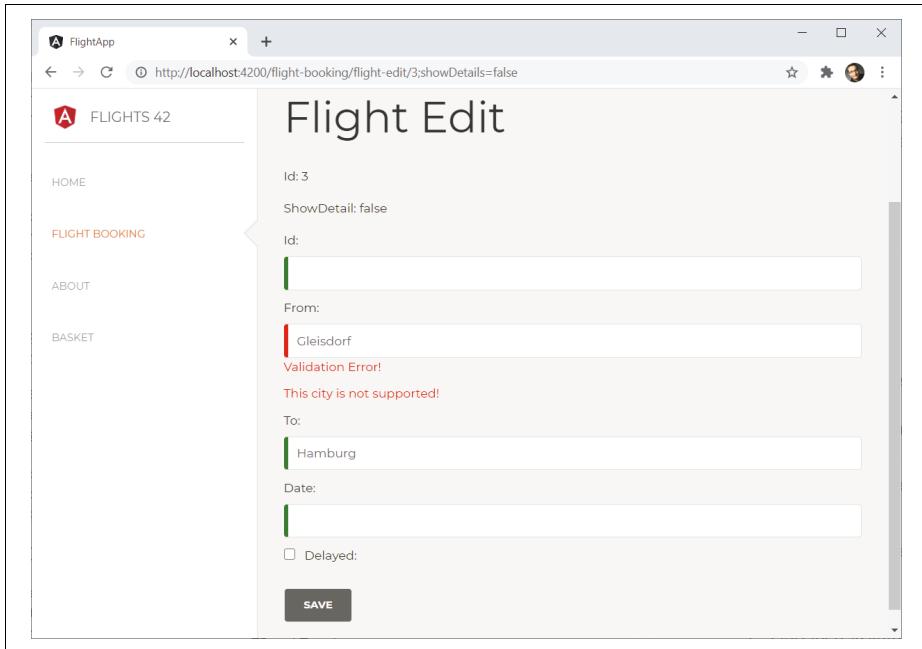


Abbildung 10-2: Validierung mit reaktivem Formular

## Parametrisierte Validatoren

Parametrisierte Validatoren für reaktive Formulare sind lediglich Factory-Funktionen, die wiederum die Validierungsfunktion zurückliefern (siehe Beispiel 10-7).

*Beispiel 10-7: Parametrisierbarer Validator für reaktive Formulare*

```
// src/app/shared/validation/reactive/city-with-params-validator.ts

import { ValidatorFn } from '@angular/forms';

export function cityWithParamsValidator(allowedCities: string[]): ValidatorFn {
  return (control) => {
    if (allowedCities.includes(control.value)) {
      return null;
    }
    return {
      city: true
    };
  };
}
```

Die Factory nimmt die Parameter entgegen und liefert die eigentliche Validierungsfunktion zurück. Diese unterliegt dem Typ `ValidatorFn` und wird somit von den reaktiven Formularen unterstützt. Da die zurückgelieferte Validierungsfunktion innerhalb der Factory definiert wird, hat sie Zugriff auf die Parameter der Factory und kann sie beim Validieren berücksichtigen.

Um den Validator zu nutzen, ist die von der Factory erhaltene Validierungsfunktion im Objektgraphen des reaktiven Formulars zu hinterlegen (siehe Beispiel 10-8).

*Beispiel 10-8: Validierungsfunktion im Objektgraphen hinterlegen*

```
// src/app/flight-booking/flight-edit/flight-edit.component.ts

[...]

// Import hinzufügen:
import { cityWithParamsValidator } from 'src/app/shared/validation/reactive/city-with-
params-validator';

[...]

this.formGroup = fb.group({
  id: [],
  from: [
    'Graz',
    [
      Validators.required,
      Validators.minLength(3),
      // Diese Funktion aufrufen:
      cityWithParamsValidator(['Graz', 'Hamburg', 'Zürich'])
    ]
  ],
  to: ['Hamburg'],
  date: [],
  delayed: []
});
[...]
```

## Asynchrone Validatoren

Auch für die asynchronen Validatoren, die Sie von den Template-getriebenen Formularen kennen, gibt es eine Entsprechung in der Welt der reaktiven Formulare.

Beispiel 10-9 veranschaulicht das mit einem parametrisierten Validator.

*Beispiel 10-9: Asynchroner Validator für reaktive Formulare*

```
// src/app/shared/validation/reactive/async-city-validator.ts

import { AsyncValidatorFn } from '@angular/forms';
import { map } from 'rxjs/operators';
import { FlightService } from 'src/app/flight-booking/flight.service';

export function asyncCityValidator(flightService: FlightService): AsyncValidatorFn {
  return (control) => {
    const error = { asyncCity: true };
    return flightService.find(control.value, '').pipe(
      map(flights => flights.length > 0 ? null : error)
    );
  };
}
```

Der von Angular angebotene Typ `AsyncValidatorFn` beschreibt asynchrone Validierungsfunktionen, die ein `control`-Objekt entgegennehmen und einen Promise oder ein Observable mit einem Fehlerbeschreibungsobjekt zurückliefern.

Die gezeigte Factory bekommt unseren `FlightService` übergeben, und die zurückgelieferte Validierungsfunktion nutzt ihn, um zu prüfen, ob für den erfassten Flughafen mindestens ein Flug vorliegt.

Die dazu aufgerufene `find`-Methode liefert ein `Observable<Flight[]>`, das `map` in ein Observable mit einem Fehlerbeschreibungsobjekt umwandelt.

Auch asynchrone Validatoren sind im Objektgraphen des reaktiven Formulars zu hinterlegen – allerdings gibt es dafür ein eigenes Argument in den `control`-Objekten sowie einen eigenen Eintrag in den Arrays, die diese beschreiben (siehe Beispiel 10-10).

*Beispiel 10-10: Asynchronen Validator registrieren*

```
// src/app/flight-booking/flight-edit/flight-edit.component.ts

[...]

// Hinzufügen:
import { asyncCityValidator } from 'src/app/shared/validation/reactive/async-city-validator';
import { FlightService } from '../flight.service';

@Component([...])
export class FlightEditComponent implements OnInit {

    [...]

    formGroup: FormGroup;

    constructor(
        private route: ActivatedRoute,
        private fb: FormBuilder,
        private flightService: FlightService) {

        this.formGroup = fb.group({
            id: [],
            from: [
                'Graz',
                [
                    Validators.required,
                    Validators.minLength(3),
                    cityWithParamsValidator(['Tripsdrill', 'Graz', 'Hamburg', 'Zürich'])
                ],
                [
                    // Hinzufügen:
                    asyncCityValidator(flightService)
                ],
            ],
            to: ['Hamburg'],
        });
    }
}
```

```

        date: [],
        delayed: []
    });

    this.formGroup.controls.delayed.statusChanges.subscribe(
        value => console.debug('delayed changed', value)
    );

    this.formGroup.statusChanges.subscribe(
        value => console.debug('whole form changed', value)
    );
}

save(): void {
    console.debug('form to save', this.formGroup.value);
    console.debug('id', this.formGroup.controls.id.value);
}
[...]
}

```

Bitte beachten Sie, dass asynchrone Validatoren im dritten Eintrag des Tupels, das das jeweilige Feld beschreibt, zu platzieren sind.

Um den asynchronen Validator mit dem `FlightService` parametrisieren zu können, lässt sich die `FlightEditComponent` diesen injizieren. Das Ergebnis des asynchronen Validators können Sie über die jeweiligen `FormControl`s ermitteln (siehe Beispiel 10-11).

Wie bei den Template-getriebenen Formularen gilt es auch hier zu berücksichtigen, dass asynchrone Validatoren nur dann ausgeführt werden, wenn kein synchroner Validator einen Fehler entdeckt. Deswegen haben wir hier die Konfiguration des synchronen um den Flughafen `Tripsdrill` erweitert. Dieser wird nun vom synchronen Validator akzeptiert, nicht jedoch vom asynchronen, zumal dafür keine Flüge vorliegen.

*Beispiel 10-11: Ergebnis des asynchronen Validators im Template verwenden*

```

<!-- src/app/flight-booking/flight-edit/flight-edit.component.html -->
[...]
<div class="error" *ngIf="formGroup.controls.from.valid === false">
    Validation Error!
</div>
<div class="error" *ngIf="formGroup.controls.from.pending">
    Validating Form ...
</div>
<div class="error" *ngIf="formGroup.controls.from.hasError('asyncCity')">
    There is no flight for this city!
</div>
[...]

```

Die Eigenschaft pending gibt darüber Auskunft, ob noch mindestens ein asynchroner Validator für die angegebene Eigenschaft ausgeführt wird. In diesem Fall weist valid den Wert undefined auf. Deswegen prüft das Beispiel auch explizit auf den Wert false.

## Multi-Field-Validatoren für reaktive Formulare

Um mehrere Felder im Rahmen der Validierung miteinander zu vergleichen, benötigen wir einen Validator für die gesamte FormGroup. Beispiel 10-12 veranschaulicht das mit einem Validator, der gegen Rundflüge prüft: Weisen from und to den gleichen Wert auf, meldet er einen Validierungsfehler.

*Beispiel 10-12: Reaktiver Multi-Field-Validator*

```
// src/app/shared/validation/reactive/round-trip-validator.ts

import { FormGroup, ValidatorFn } from '@angular/forms';

export function roundTripValidator(): ValidatorFn {
  return (control) => {
    const formGroup = control as FormGroup;
    const from = formGroup.controls.from.value;
    const to = formGroup.controls.to.value;

    if (from === to) {
      return {
        roundTrip: true
      };
    }

    return null;
  };
}
```

Da der an den Validator übergebene Parameter vom Typ AbstractControl ist, muss er mit einer Type Assertion in eine FormGroup übergeführt werden. Danach navigiert der Validator zu den FormControls für from und to und vergleicht deren Werte.

Zum Registrieren dieses Validators können Sie das zweite Argument der vom FormBuilder angebotenen group-Methode verwenden (siehe Beispiel 10-13).

*Beispiel 10-13: Registrierung eines Multi-Field-Validators*

```
// src/app/flight-booking/flight-edit/flight-edit.component.ts

[...]

// Einfügen:
import { roundTripValidator }
  from 'src/app/shared/validation/reactive/round-trip-validator';

@Component([...])
export class FlightEditComponent implements OnInit {
```

```

[...]
formGroup: FormGroup;

constructor(
  private route: ActivatedRoute,
  private fb: FormBuilder,
  private flightService: FlightService
) {

  this.formGroup = fb.group({
    id: [],
    from: [
      'Graz',
      [
        Validators.required,
        Validators.minLength(3),
        cityWithParamsValidator(['Tripsdrill', 'Graz', 'Hamburg', 'Zürich'])
      ],
      [
        asyncCityValidator(flightService)
      ]
    ],
    to: ['Hamburg'],
    date: [],
    delayed: []
  },
  // Weiteres Argument einfügen:
  {
    validators : [roundTripValidator()]
  });
}

[...]
}
[...]
}

```



Neben der Eigenschaft `validators` existiert auch eine Eigenschaft `asyncValidators` zum Registrieren asynchroner Validatoren für `FormGroups`.

Um zu prüfen, ob ein Validierungsfehler vorliegt, nutzen Sie die `hasError`-Methode der `FormGroup`:

```

<!-- src/app/flight-booking/flight-edit/flight-edit.component.html -->
[...]

<form [formGroup]="formGroup">

  <div class="error" *ngIf="formGroup.hasError('roundTrip')">
    Round Trips are not supported!
  </div>

  [...]
</form>

```

# Geschachtelte Formulare

Da Sie beim Einsatz reaktiver Formulare den Aufbau des Objektgraphen vollständig kontrollieren, können Sie auch sehr einfach Ihre Formulare untergliedern. Beispielsweise können Sie jedem Formularabschnitt eine eigene FormGroup spendieren. Das hilft, die Übersicht zu wahren. Außerdem erlaubt es, ein Formular auf verschiedene Komponenten aufzuteilen, indem einzelne FormGroups an Child-Komponenten weitergereicht werden.

## Geschachtelte FormGroups

Eine FormGroup kann nicht nur auf konkrete FormControls verweisen, sondern auch auf weitere FormGroups sowie auf FormArrays. Hier möchten wir uns Ersterem zuwenden. Unter »Wiederholgruppen mit FormArray« auf Seite 250 weiter unten behandeln wir die FormArrays.

Um FormGroups zu verschachteln, lassen Sie lediglich ein Schlüssel/Wert-Paar der äußeren FormGroup auf die innere verweisen. Beispiel 10-14 demonstriert das, indem es eine weitere FormGroup namens routeFormGroup einführt.

*Beispiel 10-14: Verschachtelte FormGroups*

```
// src/app/flight-booking/flight-edit/flight-edit.component.ts  
[...]  
  
@Component([...])  
export class FlightEditComponent implements OnInit {  
  
    [...]  
  
    formGroup: FormGroup;  
    routeFormGroup: FormGroup;  
  
    constructor(  
        private route: ActivatedRoute,  
        private fb: FormBuilder,  
        private flightService: FlightService  
    ) {  
  
        this.routeFormGroup = fb.group({  
            from: [  
                'Graz',  
                [  
                    Validators.required,  
                    Validators.minLength(3),  
                    cityWithParamsValidator(['Tripsdrill', 'Graz', 'Hamburg', 'Zürich'])  
                ],  
                [  
                    asyncCityValidator(flightService)  
                ]  
            ],  
        },  
    );  
}
```

```

        to: ['Hamburg'],
    },
{
    validators: [roundTripValidator()]
});

this.formGroup = fb.group({
    id: [],
    route: this.routeFormGroup,
    date: [],
    delayed: []
});

[...]
}

[...]
}

```

Die neue `routeFormGroup` repräsentiert die Felder `from` und `to` sowie den darauf operierenden Round-Trip-Validator. Die ursprüngliche `FormGroup` verweist darauf über den Schlüssel `route`.

Im Template verweist die Direktive `formGroupName` auf den vergebenen Schlüssel `route` (siehe Beispiel 10-15).

*Beispiel 10-15: Template mit verschachtelten FormGroups*

```

<!-- src/app/flight-booking/flight-edit/flight-edit.component.html -->

[...]

<form [formGroup]="formGroup">

    <div class="form-group">
        <label>Id:</label>
        <input formControlName="id" class="form-control">
    </div>

    <fieldset formGroupName="route">
        <!-- Alternative: [formGroup]="routeFormGroup" -->

        <div class="error" *ngIf="routeFormGroup.hasError('roundTrip')">
            Round Trips are not supported!
        </div>

        [...]

        <input formControlName="from" class="form-control">

        <div class="error" *ngIf="!routeFormGroup.controls.from.valid">
            Validation Error!
        </div>

        [...]
    
```

```

<input formControlName="to" class="form-control">
[...]
</fieldset>
[...]
</form>
```

Die auf das `fieldset` angewandte Direktive `formGroupName` sucht in der aktuellen `FormGroup` nach dem Schlüssel `route`. Alle Angaben innerhalb des `fieldset` gelten relativ zur somit gefundenen inneren `FormGroup`.

Bitte beachten Sie, dass nun die Zugriffe auf die `FormControl`s über die `routeForm` Group führen.



Als Alternative zum Einsatz von `formGroupName` können Sie auch die Direktive `formGroup` verwenden:

```
<fieldset [formGroup]="routeFormGroup">
  [...]
</fieldset>
```

In diesem Fall geben Sie jedoch anstatt eines Schlüssels, der auf einen Eintrag in der aktuellen `FormGroup` verweist, eine Datenbindung auf das konkrete `FormGroup`-Objekt in Ihrer Komponente an.

Optisch ändert diese Anpassung an unserer Anwendung nichts. Allerdings können Sie nun sehr einfach auf den Zustand der gesamten `FormGroup` `route` zugreifen. Hier aggregiert nun Angular Eigenschaften wie `valid` oder `dirty`:

```
<p *ngIf="routeFormGroup.dirty">Route changed!</p>
```

Außerdem lässt sich diese `FormGroup` an eine Subkomponente mit einem Property-Binding weiterreichen. Diese könnte sich um das Darstellen der Route kümmern:

```
<!-- Fiktive Komponente: -->
<app-flight-route [routeFormGroup]="routeFormGroup"></app-flight-route>
```

Somit können Sie ein Formular auf mehrere Komponenten aufteilen

## Wiederholgruppen mit FormArray

Der Objektgraph, den wir zum Beschreiben reaktiver Formulare verwenden, kann neben `FormControl`s und `FormGroup`s auch sogenannte `FormArray`s beinhalten. Damit lassen sich Wiederholgruppen abbilden. Um diesen Mechanismus zu demonstrieren, erweitern wir unser Formular um die Möglichkeit, pro Flug mehrere Sitzplatzkategorien zu verwalten (siehe Abbildung 10-3).

Um mehrere Sitzplatzkategorien verwalten zu können, erhält die `FlightEditComponent` ein `FormArray` namens `categoriesFormArray` (siehe Beispiel 10-16).

The screenshot shows a web browser window for 'FlightApp' at the URL <http://localhost:4200/flight-booking/flight-edit/3;showDetails=false>. The left sidebar has links for 'HOME', 'FLIGHT BOOKING' (which is active), 'ABOUT', and 'BASKET'. The main form has fields for 'To:' (Graz, Hamburg), 'Date:', and a checked 'Delayed:' checkbox. Below this, there's a section for managing seat categories. A red box highlights a 'FormArray' containing two identical row structures: 'Category Name:' and 'Base Price:'. At the bottom are 'ADD CATEGORY' and 'SAVE' buttons.

Abbildung 10-3: Verwaltung mehrerer Sitzplatzkategorien mit FormArray

Beispiel 10-16: Wiederholgruppe mit FormArray implementieren

```
// src/app/flight-booking/flight-edit/flight-edit.component.ts
```

```
[...]
```

```
// FormArray hinzufügen:
```

```
import { FormArray, FormBuilder, FormControl, FormGroup, Validators } from '@angular/forms';
```

```
[...]
```

```
@Component([...])
```

```
export class FlightEditComponent implements OnInit {
```

```
    formGroup: FormGroup;
```

```
    routeFormGroup: FormGroup;
```

```
// Hinzufügen:
```

```
    categoriesFormArray: FormArray;
```

```
    constructor(
```

```
        private route: ActivatedRoute,
```

```
        private fb: FormBuilder,
```

```
        private flightService: FlightService
```

```
    ) {
```

```

// Leeres Form-Array erzeugen:
this.categoriesFormArray = fb.array([]);

[...]

this.formGroup = fb.group({
  id: [],
  [...]
  categories: this.categoriesFormArray
});

[...]

}

addCategory(): void {
  this.categoriesFormArray.push(
    this.fb.group({
      categoryName: [],
      basePrice: []
    })
  );
}

[...]
}

```

Die FormGroup, die das gesamte Formular beschreibt, verweist auf das categories FormArray über ein Schlüssel/Wert-Paar. Die neue Methode addCategory fügt in das FormArray eine neue FormGroup ein, die eine Sitzplatzkategorie beschreibt.

Im Template referenziert einfieldset mit der Direktive formArrayName das FormArray.

#### *Beispiel 10-17: Iterieren der Einträge eines FormArray*

```

<!-- src/app/flight-booking/flight-edit/flight-edit.component.html -->
[...]

<fieldset formArrayName="categories">
  <div *ngFor="let group of categoriesFormArray.controls; let i = index"
       [formGroupName]="i" class="form-group">

    <label>Category Name:</label>
    <input formControlName="categoryName" class="form-control">
    <label>Base Price:</label>
    <input formControlName="basePrice" class="form-control">
  </div>
</fieldset>

<button class="btn btn-default" (click)="addCategory()">Add Category</button>

[...]

```

Dazu gibt `formArrayName` den Namen `categories` an. Dieser fungiert als Schlüssel innerhalb der `FormGroup`, die das gesamte Formular beschreibt. Somit beziehen sich alle Angaben innerhalb des `fieldset` auf das `FormArray`, das sich hinter diesem Schlüssel verbirgt.

Mittels `*ngFor` iteriert das `div` sämtliche `FormGroup`s innerhalb des Arrays. Der Ausdruck `let i = index` erzeugt eine Template-Variable, die den jeweiligen Array-Index erhält, also 0 für den ersten Eintrag, 1 für den zweiten Eintrag etc. Die jeweils aktuelle `FormGroup` wählt es mit der Direktive `formGroupName` aus. Als Name fungiert hier der jeweilige Array-Index.

Alle Angaben innerhalb des `div` beziehen sich also darauf. Die beiden `inputs` durchsuchen diese `FormGroup` und binden sich an die Eigenschaften `categoryName` bzw. `basePrice`.

## Dynamische Formulare

Reaktive Formulare eignen sich auch als Basis für Formulargeneratoren. Diese benötigen neben dem Objektgraphen weitere Metainformationen. Beispielsweise könnten Sie die Felder unseres Formulars wie folgt beschreiben:

```
// src/app/flight-booking/flight-edit/flight-edit.component.ts  
[...]  
@Component([...])  
export class FlightEditComponent implements OnInit {  
    [...]  
  
    metaData = [  
        { label: 'FlugNummer', name: 'id', type: 'text' },  
        { label: 'Route', name: 'route', type: 'readonly' },  
        { label: 'Date', name: 'date', type: 'text' },  
        { label: 'delayed', name: 'delayed', type: 'checkbox' }  
    ];  
    [...]  
}
```

Diese Metadaten können natürlich auch zur Laufzeit aus einer serverseitigen Konfiguration bezogen werden. Wir gehen hier außerdem davon aus, dass die bisher verwendete `FormGroup` nach wie vor existiert. Auch diese könnte anhand von Metadaten aufgebaut werden.

Das Template könnte nun die Metadaten durchlaufen und für jeden Eintrag ein Feld rendern:

```
<!-- src/app/flight-booking/flight-edit/flight-edit.component.html -->  
[...]
```

```

<h2>Dynamic Form</h2>

<form [formGroup]="formGroup">
  <div class="form-group" *ngFor="let item of metaData">
    <label>{{item.label}}:</label>
    <input *ngIf="item.type === 'text'" [formControlName]="item.name"
           class="form-control">
    <input *ngIf="item.type === 'checkbox'" [formControlName]="item.name"
           type="checkbox">
    <div *ngIf="item.type === 'readonly'">
      {{ formGroup.controls[item.name].value | json }}
    </div>
  </div>
</form>

```

Die Datenbindung erfolgt über die `formControlName`-Direktive. Sie bekommt nun statt eines fixen Werts den jeweiligen Namen aus den Metadaten zugewiesen. Um auch die `FormGroup` mit dem Namen route editierbar zu gestalten, könnten Sie hierfür an eine benutzerdefinierte Komponente delegieren.

## Zusammenfassung

Bei reaktiven Formularen ist es die Aufgabe der Entwicklerin oder des Entwicklers, mit einem Objektgraphen das Formular zu beschreiben. Daraus ergeben sich auch die Vor- und Nachteile dieser beiden Ansätze: Während Template-getriebene Formulare einfach zu nutzen sind, bieten reaktive Formulare mehr Freiheiten. Außerdem gestalten sich die Templates bei reaktiven Formularen schlanker. Das verbessert die Wartbarkeit, aber auch die Testbarkeit.

Darüber hinaus machen es reaktive Formulare auf einfache Weise möglich, Formulare zu untergliedern: Einzelne Formularabschnitte können eine eigene `FormGroup` erhalten, und Wiederholgruppen lassen sich dank `FormArrays` darstellen. Zur Verbesserung der Übersicht lassen sich sogar einzelne `FormGroup`s über Datenbindung und Child-Komponenten weitergeben. Somit können Sie ein großes Formular auf mehrere Komponenten aufteilen.

# Reactive Extensions Library for JavaScript (RxJS)

Observables sind in Angular-Anwendungen allgegenwärtig, zumal das Framework damit asynchrone Operationen repräsentiert. Solche asynchrone Operationen sind bei Browseranwendungen sehr häufig anzutreffen. Schließlich stellt uns der Browser lediglich einen Hauptthread zur Verfügung, und diesen wollen wir nicht mit lang laufenden Operationen blockieren. Genau das hätte zur Folge, dass die Anwendung einfriert.

Um das zu verhindern, bietet der Browser solche Operationen lediglich als asynchrone Operationen an. Sie blockieren den Hauptthread nicht und informieren uns, sobald sie abgeschlossen sind. Dazu rufen sie eine zuvor registrierte Funktion, einen sogenannten Callback, auf.

Leider können asynchrone Operationen und Callbacks in unübersichtlichen Code ausarten. Genau dieses Problem versuchen Observables in den Griff zu bekommen. Sie erlauben es, asynchrone Codestrecken weitestgehend linear abzubilden, sodass Sie beim Lesen nicht zwischen verschiedene Callbacks hin- und herspringen müssen. Außerdem nehmen uns Observables einige wiederkehrende Aufgaben ab.

Dieses Kapitel wirft einen strukturierten Blick auf Observables sowie auf die dahinterstehende Bibliothek *Reactive Extensions Library for JavaScript* (RxJS).

## Grundlegende Typen von RxJS

Zunächst betrachten wir ein paar grundlegende Konzepte, die wir in den folgenden Abschnitten nutzen, um anhand einer Fallstudie weiter in die Tiefen von RxJS vorzudringen.

### Observables, Observer und Operatoren

Im Gegensatz zu herkömmlichen Variablen informieren uns *Observables*, wenn neue Daten vorliegen. Dazu registrieren wir einen sogenannten *Observer* mittels *subscribe* (siehe Abbildung 11-1).

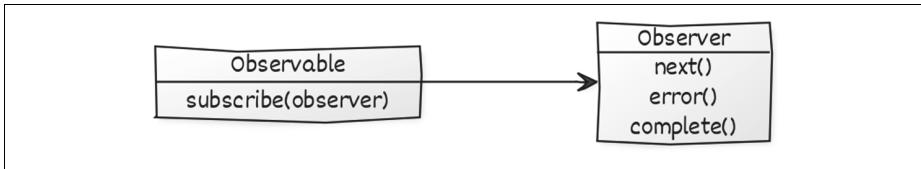


Abbildung 11-1: Observer-Muster

Das Observable wird häufig als Datenquelle oder Datenstrom gesehen. Der Observer fungiert als Datensenke und weist drei Methoden auf, die Informationen vom Observable entgegennehmen.

#### *next*

Nimmt den nächsten Wert entgegen. Wie viele Werte es insgesamt gibt, hängt von der Datenquelle des Observables ab. Repräsentiert ein Observable einen klassischen HTTP-Aufruf, liefert es wahrscheinlich nur eine einzige HTTP-Antwort zurück, zumal das dem Charakter von HTTP entspricht. Repräsentiert das Observable hingegen ein Browserevent, z.B. click, stößt das Observable die Methode next bei jedem Auftreten dieses Events an.

#### *error*

Mit dieser Methode informiert das Observable den Observer über einen auftretenen Fehler. Danach veröffentlicht das Observable per definitionem keine weiteren Werte mehr. Deswegen sollten Fehler wie weiter unten beschrieben behandelt werden.

#### *complete*

Diese Methode gibt an, dass das Observable keine weiteren Werte mehr senden und deswegen nun geschlossen wird.

Alle drei Methoden sind optional. Benötigen Sie zum Beispiel keine Benachrichtigung beim Schließen, können Sie complete weglassen.

Um Daten im Verlauf des Datenflusses zu manipulieren, kommen sogenannte Operatoren zum Einsatz. Ein erstes Experiment, das das Zusammenspiel von Observable, Observer und Operatoren demonstriert, haben wir in der HomeComponent platziert (siehe Beispiel 11-1).

#### *Beispiel 11-1: HttpClientObservable*

```
// src/app/home/home.component.ts

import { map } from 'rxjs/operators';
[...]
constructor(private http: HttpClient) { }

[...]

this.http.get<Flight>('http://www.angular.at/api/flight/3')
  .pipe(
    map(
```

```

        map(flight => new Date(flight.date)),
        // Hier könnten weitere Operatoren stehen.
    )
    .subscribe({
        next: value => console.log('date', value),
        error: err => console.error('error', err),
        complete: () => console.log('complete')
    });

```

Der HttpClient fragt beim Backend einen Flug an und repräsentiert die Antwort als Observable<Flight>. Die Methode pipe nimmt einen oder mehrere Operatoren entgegen. Dabei handelt es sich um Funktionen, die häufig aus dem Namensraum rxjs/operators stammen. Der Operator map bildet den aktuellen Wert auf einen anderen ab. In unserem Fall bildet er den Flug auf sein Datum ab. Dieses Datum, das zunächst als ISO-String vorliegt, wandelt das Beispiel in ein Date-Objekt um. Die Methode subscribe registriert anschließend den Observer mit den genannten drei Methoden.



Anstelle eines Observers nimmt subscribe auch eine Funktion entgegen. Dieser sogenannte next-Handler erhält wie die Methode next den jeweils nächsten Wert:

```

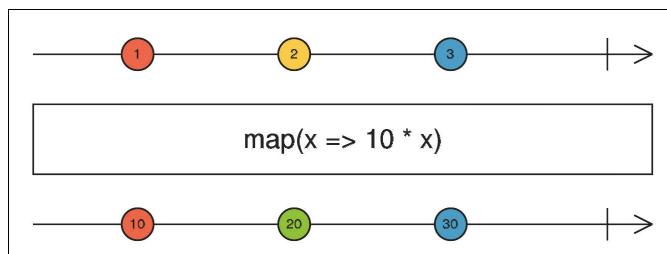
this.http.get<Flight>('http://www.angular.at/api/flight/3')
    .pipe(map(flight => new Date(flight.date)))
    .subscribe(value => console.log('date', value));

```

Früher konnte man neben einem next-Handler auch zwei weitere Handler, die den Methoden error und complete des Observers entsprachen, übergeben. Diese Option ist mittlerweile als veraltet (*deprecated*) gekennzeichnet und wird in einer der nächsten RxJS-Versionen entfernt.



Eine sehr eingängige Art der Visualisierung von Operatoren stellen Marble-Diagramme dar. Die Funktionsweise von map lässt sich damit zum Beispiel wie folgt darstellen:



Sowohl die offizielle RxJS-Dokumentation (<https://rxjs-dev.firebaseio.com>), aus der die Marble-Diagramme in diesem Kapitel entnommen wurden, als auch zahlreiche Community-Bestrebungen nutzen diese Darstellungsform. Hervorzuheben ist hier die Seite <https://xmarbles.com>, die für ausgewählte Operatoren interaktive Marble-Diagramme bietet.

## Observables instanziieren

Bis jetzt haben wir lediglich von bestehenden Funktionen bzw. Methoden, wie zum Beispiel von jenen des `HttpClient`, Observables erhalten. Zur Schaffung eines besseren Verständnisses wollen wir nun eigene Observable instanziieren. Dabei sei aber erwähnt, dass es für viele Zwecke bereits Funktionen zum Erzeugen von Observables gibt. Weiter unten zeigen wir ein paar Exemplare dafür.

Unser erstes selbst instanzierte Observable soll einfach zwei Werte veröffentlichen und danach schließen (siehe Beispiel 11-2).

*Beispiel 11-2: Instanziierung eines einfachen Observables*

```
// src/app/shared/observable-factories.ts

import { Observable } from 'rxjs';

// eslint-disable-next-line prefer-arrow/prefer-arrow-functions
export function simpleObservable(): Observable<number> {
    return new Observable<number>(observer => {

        observer.next(4711);
        observer.next(815);
        // observer.error('Manfred braucht einen Kaffee!');
        observer.complete();

        return () => {
            console.log(`teardown!`);
        };
    });
}
```

Die Factory `simpleObservable` ruft lediglich den Konstruktor des Observables mit dem Schlüsselwort `new` auf. Der übergebene Lambda-Ausdruck erhält den Observer übergeben, sobald er sich mit `subscribe` anmeldet. Diese Information lässt schon vermuten, dass prinzipiell eine 1:1-Beziehung zwischen einem Observable und einem Observer existiert. Der Abschnitt »Multicasting« auf Seite 279 geht weiter unten im Detail darauf ein.

Das Beispiel versendet zwei Werte mit `next` und schließt danach mit `complete`. Das Auslösen eines Fehlers deutet es mit einem Kommentar an. Beim am Ende zurückgelieferten Lambda-Ausdruck handelt es sich um eine Teardown-Funktion. Diese führt das Observable aus, wenn sich der Observer abmeldet. Das ist auch der Fall, wenn das Observable selbst `complete` aufruft. Typischerweise ist diese Funktion für Aufräumarbeiten zuständig.

Die Teardown-Funktion ist übrigens optional. Gibt es keine Aufräumarbeiten, liefern sie in der Regel keinen Wert zurück.

Zur Demonstration rufen wir die Funktion `simpleObservable` in der `HomeComponent` auf und registrieren einen Observer (siehe Beispiel 11-3).

*Beispiel 11-3: Die Hilfsfunktion simpleObservable aufrufen.*

```
// src/app/home/home.component.ts
[...]

// Hinzufügen:
import { simpleObservable } from '../shared/observable-factories';

[...]

// Zu Konstruktor hinzufügen:
const simple$ = simpleObservable();
simple$.subscribe({
  next: value => console.log('next', value),
  error: err => console.error('error', err),
  complete: () => console.log('complete')
});
```



Um Observables auf den ersten Blick zu erkennen, hat sich die Nutzung des Suffixes \$ eingebürgert. Deswegen heißt unser Observable `simple$` und nicht nur `simple`.

Als Resultat sollten wir nun die Ausgaben `next 4711`, `next 815`, `complete` und `teardown!` in der JavaScript-Konsole des Browsers sehen.

Auch wenn dieses Beispiel schon einige Konzepte von RxJS in Aktion zeigt, ist die Veröffentlichung bereits vorliegender Werte ein eher seltener Anwendungsfall: Liegen Werte wie 4711 und 815 synchron vor, könnten sie auch mit einfacheren Datenstrukturen wie z.B. Arrays verwaltet werden. Deswegen geht das Beispiel in Beispiel 11-4 einen Schritt weiter und veröffentlicht asynchrone Werte.

*Beispiel 11-4: Observable mit Intervall von drei Zahlen*

```
// src/app/shared/observable-factories.ts

import { Observable } from 'rxjs';

[...]

export function simpleInterval(): Observable<number> {
  return new Observable<number>(observer => {
    let counter = 0;

    const handle = setInterval(() => {
      observer.next(counter);
      // observer.error('Habe mich verzählt ...')

      counter++;
      if (counter >= 3) {
        observer.complete();
      }
    }, 1000);
  });
}
```

```

        return () => {
            clearInterval(handle);
            console.log(`teardown!`);
        };
    });
}

```

Das von `simpleInterval` gelieferte Observable liefert die Werte 0, 1 und 2 mit einem zeitlichen Abstand von jeweils 1.000 Millisekunden. Hierzu stützt sie sich auf die von JavaScript ab Werk gelieferte Funktion `setInterval`. Das damit definierte Intervall schließt die Teardown-Funktion mit der ebenfalls von JavaScript gebotenen Funktion `clearInterval`.

Zum Ausprobieren von `simpleInterval` erweitern wir wieder die `HomeComponent`.

*Beispiel 11-5: Die Hilfsfunktion simpleInterval aufrufen*

```
// src/app/home/home.component.ts

import { simpleInterval } from '../shared/observable-factories';

[...]

const interval$ = simpleInterval();
const sub = interval$.subscribe({
    next: value => console.log('next', value),
    error: err => console.error('error', err),
    complete: () => console.log('complete')
});

setTimeout(() => sub.unsubscribe(), 2500);
```

Im Wesentlichen entspricht dieses Beispiel dem oben gezeigten. Allerdings nimmt es das von `subscribe` zurückgelieferte Subscription-Objekt entgegen. Dieses Objekt hat nur einen einzigen Zweck – das Abmelden des Observers mittels `unsubscribe`. Da das Beispiel `unsubscribe` bereits nach 2.500 Millisekunden aufruft, liefert das Observable lediglich die Werte 0 und 1, bevor es schließt und die Methode `complete` sowie die Teardown-Funktion anstößt.

Die Ausgabe auf der JavaScript-Konsole gestaltet sich also wie folgt: `next 0, next 1` und `teardown!`. Würden wir `unsubscribe` hingegen nicht aufrufen, wäre die Ausgabe `next 0, next 1, next 2, complete` und `teardown!`.

Im Zweifelsfall sollten Sie die Methode `unsubscribe` aufrufen, wenn Sie keine weiteren Daten mehr von einem Observable benötigen. Ansonsten kann sich ein Memory-Leak ergeben, wenn das Observable nicht von selbst schließt. Ob und wann ein Observable sich selbst schließt, hängt von der Art des Observables ab. Die vom `HttpClient` gelieferten Observables schließen zum Beispiel unmittelbar nach Empfang der Antwortnachricht. Ein Observable, das einen Event-Handler repräsentiert, schließt hingegen nicht automatisch, zumal jedes Event noch mindestens ein (weiteres) Mal angestoßen werden kann.

## Subjects

Bei Subjects handelt es sich um eine besondere Art von Observables. Während ein klassisches Observable selbst für das Veröffentlichen seiner Werte verantwortlich ist, erhält sie ein Subject von außerhalb (siehe Abbildung 11-2).

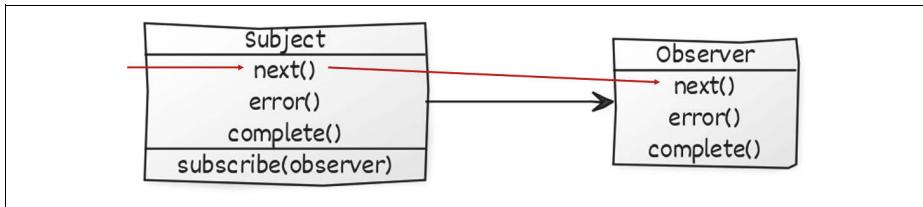


Abbildung 11-2: Subject und Observer

Technisch gesehen, vereint ein Subject die Rollen eines Observers, der sich über `next`, `error` und `complete` informieren lässt, und eines Observables, das diese Informationen veröffentlicht:

```
import { Subject } from 'rxjs';
[...]
const subject = new Subject<string>();
subject.next('A');
subject.subscribe(value => console.debug('Subject says', value));
subject.next('B');
subject.next('C');
```

Dieses einfache Beispiel gibt lediglich die Werte B und C aus. Das liegt daran, dass die Anmeldung mit `subscribe` erst nach dem Veröffentlichen von A erfolgt. Man spricht hier auch vom *Late-Subscriber-Problem*. Eine Lösung für dieses Problem besteht daran, Subjects mit einem Gedächtnis auszustatten. Das sogenannte `BehaviorSubject` merkt sich beispielsweise den letzten Wert:

```
import { BehaviorSubject } from 'rxjs';
[...]
const subject = new BehaviorSubject<string>('INIT');
subject.next('A');
subject.subscribe(value => console.debug('Subject says', value));
subject.next('B');
subject.next('C');
```

Deswegen findet man trotz der späten Anmeldung nun die Werte A, B und C auf der JavaScript-Konsole. Außerdem hat ein `BehaviorSubject` immer einen Initialwert (hier `INIT`), den es ausliefert, wenn noch kein weiterer Wert mittels `next` festgelegt wurde.

Während sich das `BehaviorSubject` nur den letzten Wert merkt, merkt sich das `ReplaySubject` mehrere veröffentlichte Werte:

```

import { ReplaySubject } from 'rxjs';

[...]

const subject = new ReplaySubject<string>(10);
subject.next('A');
subject.next('B');
subject.subscribe(value => console.debug('Subject says', value));
subject.next('C');

```

Dieses ReplaySubject merkt sich die letzten zehn Werte. Deswegen ergibt sich auch hier die Ausgabe A, B und C.

Es gehört übrigens nicht zum guten Ton, anderen Objekten direkten Zugriff auf ein Subject zu geben. Das würde es anderen Systembestandteilen erlauben, Werte in unserem Namen zu veröffentlichen. Verwirrung ist hier vorprogrammiert. Statt dessen ist es üblich, Subjects für die Nutzung durch Dritte in Observables umzuwandeln. Dazu bietet die Klasse Subject die Methode asObservable:

```

const sender = new BehaviorSubject<string>('init');
const receiver$ = sender.asObservable();
receiver$.subscribe(value => console.debug('Subject says', value));
sender.next('A');
sender.next('B');
sender.next('C');

```

Das Subject selbst behalten wir uns in der Rolle des Senders, während das daraus abgeleitete Observable für Dritte, die nur in der Rolle des Empfängers auftreten, bestimmt ist.

## Observables vs. Promises

Bei ersten Schritten mit Observables ergibt sich häufig die Frage, wie sich diese von den mittlerweile im ECMAScript-Standard verankerten Promises (siehe Kapitel 2) unterscheiden. Darauf gibt es gleich mehrere Antworten:

- Promises liefern nur einen Wert zurück, Observables können hingegen mehrere Werte im Verlauf der Zeit liefern.
- Observables können durch den Aufruf von unsubscribe abgebrochen werden. Bei Promises existiert diese Möglichkeit nicht.
- Observables können dank Operatoren Daten im Zuge des Flusses sehr flexibel manipulieren.
- Promises können mit den modernen Schlüsselwörtern async/await (siehe Kapitel 2) verwendet werden.

Es spricht nichts dagegen, Promises zu nutzen, wenn Sie damit auskommen. Auf der anderen Seite lässt sich auch häufig beobachten, dass zugunsten der Einheitlichkeit ausschließlich Observables zum Einsatz kommen.

Sie können Promises auch in Observables und vice versa umwandeln. Die nachfolgenden Abschnitte gehen darauf ein.

## Observables in Promises umwandeln

Zum Umwandeln von Observables in Promises war es bis RxJS-Version 6 üblich, die Methode `toPromise` zu nutzen:

```
const promise = interval(500).pipe(take(4)).toPromise();
promise.then(v => console.debug('promise', v));
```

Man mag es zwar nicht auf den ersten Blick erkennen, aber dieses Beispiel liefert nach 2 Sekunden (4 \* 500 Millisekunden) den Wert 3. Das liegt daran, dass `toPromise` wartet, bis das Observable geschlossen wurde und im Zuge des Complete-Events den letzten veröffentlichten Wert liefert.

Das bedeutet, dass der Promise nur den letzten Wert bekommt, obwohl das Observable im Intervall von jeweils 500 Millisekunden die Werte 0, 1, 2 und 3 liefert. Ein solcher Kunstgriff ist notwendig, zumal ein Promise nur einen einzigen Wert liefern kann. Da dies Verwirrung stiften kann, wurde `toPromise` mit RxJS 7, das gemeinsam ab Angular 13 ausgeliefert wird, als veraltet (*deprecated*) gekennzeichnet. An seine Stelle treten ab RxJS 7 die Funktionen `firstValueFrom` und `lastValueFrom`. Diese Funktionen haben den Vorteil, dass ihre Namen darüber Auskunft geben, ob sie den ersten oder den letzten Wert des Observables über den Promise veröffentlichen:

```
import { firstValueFrom, lastValueFrom } from 'rxjs';

const promise = lastValueFrom(interval(500).pipe(take(4)));
promise.then(v => console.debug('promise', v));

const otherPromise = firstValueFrom(interval(500).pipe(take(4)));
otherPromise.then(v => console.debug('promise', v));
```

## Promises in Observables umwandeln

Falls Sie durchgängig reaktiv programmieren wollen, werden Sie wahrscheinlich häufiger Promises von existierenden APIs in Observables abbilden müssen. Diese Aufgabe können Sie mit der Funktion `from` aus dem Namensraum `rxjs` durchführen (siehe Beispiel 11-6).

*Beispiel 11-6: Promise in Observable umwandeln*

```
function timerAsPromise(time: number): Promise<number> {
  return new Promise<number>((resolve, reject) => {

    if (time < 0) {
      reject();
    }

    setTimeout(() => resolve(time), time);
  });
}
```

```
}
```

```
from(timerAsPromise(500)).pipe(
  map(msec => msec / 1000)
)
.subscribe(v => console.debug('sec', v));
```

Solche Observables schließen automatisch, nachdem sie das Ergebnis des Promises veröffentlicht haben.

## Gruppen von Operatoren

Operatoren können, wie eingangs gezeigt, den Datenfluss beeinflussen. Daneben zählt man auch Factories, die neue Observables erzeugen, zu den Operatoren. Insgesamt bringt RxJS über 100 solcher Funktionen. Die wenigsten Entwickler und Entwicklerinnen kennen alle Operatoren im Detail, und in vielen Fällen kommt man auch mit weniger als einem Dutzend aus. Um die vielen Operatoren jedoch besser einordnen zu können, ist es nützlich, zu wissen, dass sich diese in verschiedene Kategorien unterteilen lassen. Hier gehen wir auf diese Kategorien genauer ein und zeigen die wichtigsten ihrer Operatoren auf. Diese Operatoren kommen in der Fallstudie, um die sich die restlichen Abschnitte dieses Kapitels dreht, zum Einsatz.

### Creation Operators

Die *Creation Operators* nehmen eine Sonderrolle ein. Sie sind die einzigen Operatoren, die nicht innerhalb von pipe zur Ausführung kommen und sich im Namensraum rxjs befinden. Alle anderen finden sich unter rxjs/operators. Es handelt sich dabei um Factories zum Erzeugen neuer Observables.

Die nachfolgende Auflistung zeigt ein paar übliche Vertreter:

*of*

Erzeugt für einen bestehenden Wert ein Observable. Dieser Wert wird an den Observer sofort nach dessen Registrierung geliefert. Diese Methode ist nützlich, um Dummy-Observables zum Testen zu erzeugen oder wenn ein zu implementierendes Interface die Nutzung eines Observables erzwingt.

*from*

Wandelt Auflistungen wie Arrays in ein Observable um. Die Einträge werden hintereinander veröffentlicht.

*throwError*

Wie of, jedoch wird ein Fehler festgelegt, den der Observer unmittelbar nach der Anmeldung erhält.

*interval*

Liefert aufsteigende Werte in einem festgelegten Intervall.

Beispiele für diese Operatoren finden sich in Beispiel 11-7.

*Beispiel 11-7: Ausgewählte Creation Operators*

```
import { of, throwError, interval } from 'rxjs';
[...]
of(4711).subscribe(v => console.log(v));
// Result: 4711

throwError('Need Coffee!').subscribe({
  next: (next) => console.error('next', next),
  error: (err) => console.error('err', err),
  complete: () => console.log('complete')
});
// Result: err Need Coffee!

interval(1000).pipe(take(5)).subscribe(v => console.log(v));
// Result: 0, 1, 2, 3, 4 in an interval of 1000ms

from([1, 2, 3]).subscribe(v => console.log('next', v));
// Result: next 1, next 2, next 3
```

## Transformation Operators

Die *Transformation Operators* transformieren einen Wert im Zuge des Datenflusses. Der wohl bekannteste Transformation Operator ist der bereits weiter oben vorgestellte `map`-Operator. Aber auch die sogenannten *Flattening Operators* wie `switchMap` oder `mergeMap`, die wir im Abschnitt »Reaktiver Entwurf« auf Seite 267 genauer behandeln, zählen dazu.

## Filtering Operators

*Filtering Operators* können ausgewählte Werte verwerfen, sodass sie nicht weiter zum Observer fließen:

### *take*

Nimmt die ersten n Werte und schließt das Observable danach.

### *filter*

Filtert die Werte anhand eines übergebenen Ausdrucks.

### *distinctUntilChanged*

Filtert Duplikate, die in Folge auftreten.

### *debounceTime*

Verwirft einen Wert, wenn innerhalb einer definierten Zeitspanne ein weiterer Wert auftritt. Diesen Operator setzt man zum Beispiel ein, wenn Tastenschläge zu einer Aktion führen sollen, sobald der Benutzer eine Pause von 300 Millisekunden einlegt.

Beispiele für diese Operatoren finden sich in Beispiel 11-8.

### Beispiel 11-8: Ausgewählte Filtering Operators

```
import { debounceTime, distinctUntilChanged, filter, map, take } from 'rxjs/operators';
[...]
interval(200).pipe(
  take(5),
  filter(v => v % 2 === 0)
)
.subscribe(v => console.log(v));
// Result: 0, 2, 4

from([1, 1, 2, 2, 1]).pipe(
  distinctUntilChanged()
)
.subscribe(v => console.log(v));
// Result: 1, 2, 1

interval(200).pipe(
  take(5),
  debounceTime(300)
)
.subscribe(v => console.log(v));
// Result: 4
```

Das letzte Beispiel mit `debounceTime` verdient eine kurze Anmerkung: Da es im Intervall von 200 Millisekunden einen neuen Wert veröffentlicht, die an `debounceTime` übergebene Zeitspanne jedoch 300 Millisekunden beträgt, werden alle bis auf den letzten Wert verworfen.

## Join Operators

*Join Operators* verbinden bestehende Observables. Ein Beispiel dafür ist `merge`, das sämtliche Werte verschiedener Observables in ein gemeinsames überführt:

```
import { merge } from 'rxjs';
[...]
merge(
  from([1, 2, 3]),
  from([4, 5, 6])
)
.subscribe(v => console.log(v));
// Ausgabe: 1, 2, 3, 4, 5, 6
```

Im Abschnitt »Datenflüsse kombinieren« auf Seite 272 gehen wir anhand unserer Beispielanwendung auf ein paar häufig benötigte Join Operators ein.



Viele Join Operators, wie auch der hier gezeigte Operator `merge`, sind gleichzeitig Factories und somit Creation Operators. Solche Operatoren nennen sich deswegen auch *Join Creation Operators*. Wie die anderen Creation Operators befinden sie sich im Namensraum `rxjs` und nicht – wie die anderen Operatoren – in `rxjs/operators`.

## Error Handling Operators

*Error Handling Operators* erlauben das Behandeln von Fehlern. Der Abschnitt »Fehlerbehandlung« auf Seite 282 widmet sich diesem Thema anhand eines Fallbeispiels.

## Multicasting Operators

Standardmäßig herrscht zwischen Observable und Observer eine 1:1-Verbindung. Möchte man die Werte eines Observables an mehrere Observer verteilen, kommen *Multicasting Operators* zum Einsatz. Wir greifen dieses Thema im Rahmen unseres Fallbeispiels im Abschnitt »Multicasting« auf Seite 279 auf.

## Utility Operators

*Utility Operators* sind Operatoren, die sich keiner anderen Gruppe wirklich zuordnen lassen:

*delay*

Verzögert jeden einzelnen Wert um eine angegebene Zeitspanne.

*tap*

Sieht vorbeifließende Werte, ohne sie zu verändern. Kommt häufig beim Debuggen zum Einsatz.

Das nachfolgende Beispiel nutzt *delay*, um nach jedem Wert eine Pause von 2.000 Millisekunden einzulegen. Bevor der Operator ungerade Werte aus dem Verkehr zieht, werden sie protokolliert.

```
from([1,2,3,4,5]).pipe(  
  delay(2000),  
  tap(v => console.debug('before filter', v)),  
  filter(v => v % 2 === 0)  
)  
.subscribe(v => console.log(v));  
  
Ergebnis: before filter 1, before filter 2, 2, before filter 3, before filter 4,  
4, before filter 5
```

## Reaktiver Entwurf

Nachdem wir uns in den letzten Abschnitten die grundlegenden Konzepte von RxJS angesehen haben, wird es nun Zeit für ein Fallbeispiel, das diese Konzepte in Aktion zeigt. Dieses Fallbeispiel wollen wir hier methodisch entwerfen und umsetzen. Dazu arbeiten wir drei Aufgaben ab, die immer zu einer reaktiven Lösung führen.

Bei unserem Beispiel handelt es sich um eine reaktive Flugsuche (siehe Abbildung 11-3).

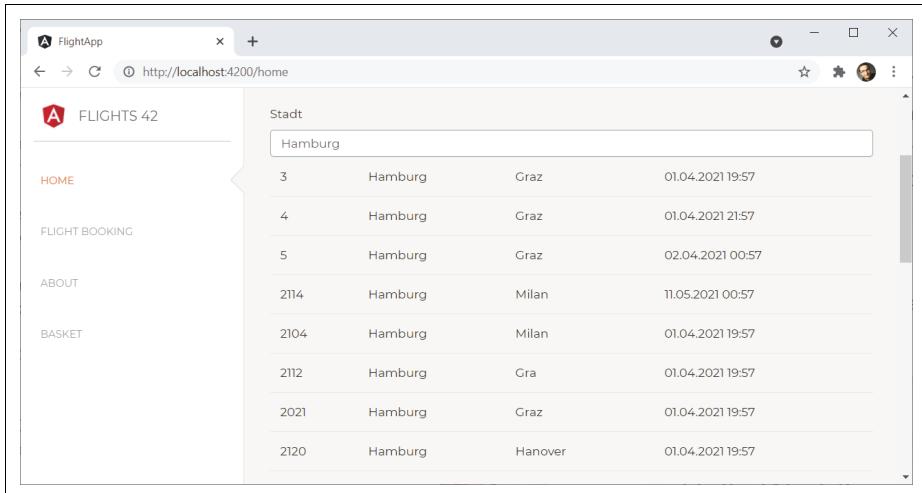


Abbildung 11-3: Reaktive Flugsuche

Dieses Beispiel ist bewusst einfach gehalten, aber gleichzeitig komplex genug, um sämtliche Aspekte dieses Kapitels zu veranschaulichen.

Zur Vereinfachung legen wir dafür eine Komponente im AppModule an:

```
ng generate component flight-typeahead
```

Um den Benutzer mit dieser neuen Komponente begrüßen zu können, platzieren wir sie in der HomeComponent:

```
<!-- src/app/home/home.component.html -->
<h1>Welcome!</h1>

<app-flight-typeahead></app-flight-typeahead>
```

Unser Beispiel wird reaktive Formulare nutzen. Deswegen ist das ReactiveFormsModuleModule in das AppModule zu importieren:

```
// src/app/app.module.ts

[...]
// Hinzufügen:
import { ReactiveFormsModule } from '@angular/forms';

@NgModule({
  imports: [
    [...]
    // Hinzufügen:
    ReactiveFormsModule,
  ],
  [...]
})
export class AppModule { }
```

Um nun methodisch zu einem reaktiven Entwurf zu gelangen, orientieren wir uns an den folgenden drei Aufgaben:

1. Datenquellen als Observables darstellen.
2. Datensenken als Observables darstellen.
3. Datenquellen mit Operatoren verändern, damit sich daraus die Datensenken ergeben.

Als Datenquelle fungiert das gezeigte Eingabefeld. In der Komponente repräsentieren wir es mit einer Instanz von FormControl (siehe Beispiel 11-9).

*Beispiel 11-9: Erste Implementierung einer reaktiven Flugsuche*

```
// src/app/flight-typeahead/flight-typeahead.component.ts

import { Component, OnInit } from '@angular/core';
import { FormControl } from '@angular/forms';
import { Observable } from 'rxjs';
import { debounceTime, switchMap } from 'rxjs/operators';
import { Flight } from '../flight-booking/flight';
import { FlightService } from '../flight-booking/flight.service';

@Component({
  selector: 'app-flight-typeahead',
  templateUrl: './flight-typeahead.component.html'
})
export class FlightTypeaheadComponent implements OnInit {

  // Quellen
  control = new FormControl();
  input$ = this.control.valueChanges;

  // Ziel
  flights$: Observable<Flight[]>;

  constructor(private flightService: FlightService) {
    this.flights$ = this.input$.pipe(
      debounceTime(300),
      switchMap(input => this.load(input))
    );
  }

  // noch eine Quelle
  load(from: string): Observable<Flight[]> {
    return this.flightService.find(from, '');
  }

  ngOnInit(): void {
  }
}
```

Das FormControl informiert uns über Benutzereingaben mit seinem Observable valueChanges, das wir input\$ zuweisen.

Eine weitere Datenquelle ist das Backend, das das gezeigte Beispiel mit seiner Methode `load` aufruft. Sie delegiert vorerst lediglich an den `FlightService`, wird jedoch im Laufe dieses Kapitels noch weitere Aufgaben bekommen.

Als Datensenke fungiert ein `Observable<Flight[]>` mit dem Namen `flights$`. Der Konstruktor leitet diese aus den Datenquellen ab. Um nicht unmittelbar auf jeden von `input$` gemeldeten Tastendruck reagieren zu müssen, wartet der Konstruktor mit `debounceTime`, bis der Benutzer eine Pause von 300 Millisekunden einlegt.

Danach bildet er die aktuelle Eingabe auf das Ergebnis von `load` ab. Dazu kommt `switchMap` zum Einsatz. Dieser Operator ist notwendig, weil `load` ein weiteres Observable zurückliefert. Käme eine klassische `map` zum Einsatz, wäre das Ergebnis ein `Observable<Observable<Flight[]>>`. Solche Konstrukte sind schwer zu verstehen, und deswegen gilt es sie so gut wie immer zu vermeiden.

Die Lösung sind sogenannte Flattening Operators (siehe »Flattening« auf Seite 271) wie `switchMap`. Sie »drücken das Observable flach«. Das bedeutet, dass sie das Ergebnis des inneren Observables in das äußere transportieren. In unserem Fall erhalten wir somit ein `Observable<Flight[]>`, und das ist genau das, was wir für `flights$` benötigen.

Das Template bindet ein Eingabefeld mit der Direktive `FormControl` an unsere `control`-Eigenschaft (siehe Beispiel 11-10).

*Beispiel 11-10: Template für die reaktive Flugsuche*

```
<!-- src/app/flight-typeahead/flight-typeahead.component.html -->

<h2 class="title">Typeahead</h2>

<div class="control-group">
  <label>Stadt</label>
  <input [FormControl]="control" class="form-control">
</div>

<table class="table table-striped">
  <tr *ngFor="let f of flights$ | async">
    <td>{{f.id}}</td>
    <td>{{f.from}}</td>
    <td>{{f.to}}</td>
    <td>{{f.date | date:'dd.MM.yyyy HH:mm'}}</td>
  </tr>
</table>
```



Während wir in Kapitel 10 gesamte `FormGroup`s gebunden haben, binden wir hier zur Vereinfachung lediglich ein einzelnes `FormControl`-Objekt.

Um die abgerufenen Flüge iterieren zu können, binden wir das Observable `flights$` mit der `async`-Pipe. Diese Pipe informiert Angular, wenn das Observable neue Werte liefert. Unter der Motorhaube meldet es dazu einen Observer an (`subscribe`) bzw., wenn es nicht mehr benötigt wird, automatisch wieder ab (`unsubscribe`).

# Flattening

Im letzten Abschnitt haben wir bereits mit `switchMap` einen sogenannten *Flattening Operator* verwendet. Solche Operatoren verhindern verschachtelte Observables. In unserem Beispiel drücken wir damit ein `Observable<Observable<Flight[]>>` flach, sodass sich ein `Observable<Flight[]>` ergibt.

Zusammen mit `switchMap` existieren insgesamt vier solcher Flattening Operators. Auf den ersten Blick machen sie alle das Gleiche. Sieht man allerdings genauer hin, stellt man fest, dass sich ihr Verhalten unterscheidet, wenn gleichzeitig mehrere Aufgaben anstehen.

## `switchMap`

Stehen mehrere Aufgaben zeitgleich an, kümmert sich `switchMap` nur um die letzte. Eventuelle andere bereits gestartete Aktionen werden abgebrochen. Dieses Verhalten bietet sich für Suchanfragen an. Korrigiert der Benutzer seine Eingabe beispielsweise von Wien auf Berlin, müssen nur noch Berlin-Flüge geladen werden. Die Suche nach Wien-Flügen kann abgebrochen werden.

## `mergeMap`

Der Operator `mergeMap` kümmert sich um alle anstehenden Aufgaben. Er parallelisiert asynchrone Operationen.

## `concatMap`

Auch `concatMap` kümmert sich um alle anstehenden Aufgaben. Asynchrone Operationen parallelisiert er jedoch nicht, sondern führt sie sequenziell aus.

## `exhaustMap`

Stehen mehrere Aufgaben gleichzeitig an, kümmert sich `exhaustMap` nur um die erste. Dieses Verhalten bietet sich bei Änderungen an: Klickt der Benutzer beispielsweise mehrfach unmittelbar hintereinander auf denselben Speichern-Button, ergibt es Sinn, die erste angeforderte Speicherung fortzuführen und die unnötigen zusätzlichen Klicks zu ignorieren.

Um diese feinen Unterschiede zu erleben, bietet sich ein kleines Experiment mit unserem Beispiel an: Verzögern Sie das Laden der Flüge mit dem Operator `delay`, um zu beobachten, wie sich diese Operatoren verhalten, wenn mehrere Anfragen gleichzeitig anstehen (siehe Beispiel 11-11).

### Beispiel 11-11: Flattening Operators ausprobieren

```
// src/app/flight-typeahead/flight-typeahead.component.ts

// Hinzufügen:
import { delay } from 'rxjs/operators';
import { mergeMap } from 'rxjs/operators';

@Component({
  selector: 'app-flight-typeahead',
  templateUrl: './flight-typeahead.component.html'
})
```

```

export class FlightTypeaheadComponent implements OnInit {

    // Quellen
    control = new FormControl();
    input$ = this.control.valueChanges;

    // Ziel
    flights$: Observable<Flight[]>;

    constructor(private flightService: FlightService) {
        this.flights$ = this.input$.pipe(
            debounceTime(300),
            // Flattening Operators ausprobieren:
            mergeMap(input => this.load(input))
        );
    }

    // noch eine Quelle
    load(from: string): Observable<Flight[]> {
        console.log('from', from);
        return this.flightService.find(from, '').pipe(
            // Delay einfügen:
            delay(7000)
        );
    }

    ngOnInit(): void {
    }
}

```

Wiederholen Sie dieses Experiment für jeden Flattening Operator. Erfassen Sie dabei mehrere Suchfilter, die Sie jeweils nach dem Debouncing von 300 Millisekunden auf einen anderen Wert – zum Beispiel von Wien auf Berlin – abändern.

Am Ende sollten Sie das `delay` wieder entfernen und auf `switchMap` zurückwechseln.

## Datenflüsse kombinieren

Es kommt immer wieder vor, dass Sie verschiedene Datenflüsse und somit verschiedene Observables kombinieren müssen. Hierzu bietet RxJS einige sogenannte *Join Operators*.

### Der Operator `combineLatest`

Der wohl am häufigsten benötigte Join Operator ist `combineLatest`. Er kombiniert jeweils den letzten Wert von zwei oder mehreren Observables zu einem Tupel (siehe Abbildung 11-4).

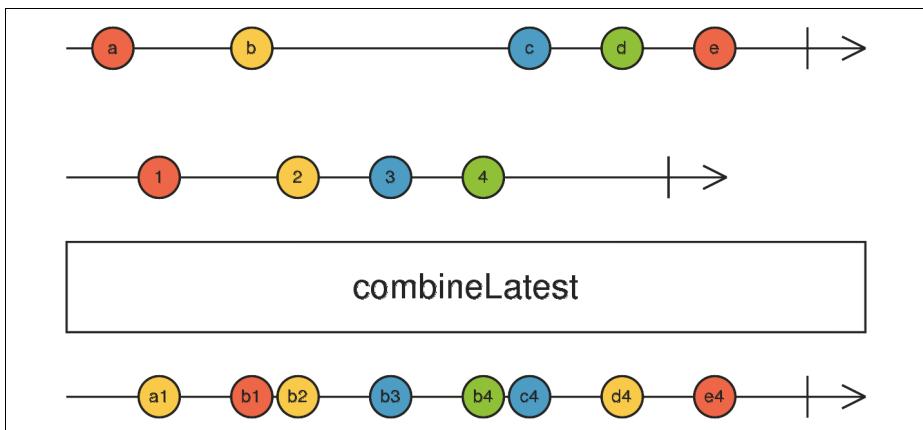


Abbildung 11-4: Marble-Diagramm für `combineLatest` (entnommen aus der offiziellen RxJS-Dokumentation)

Zur Veranschaulichung erweitern wir die `FlightTypeaheadComponent` um eine Datenquelle `online$`, die einen wechselnden Onlinestatus unserer Anwendung simuliert (siehe Beispiel 11-12).

#### Beispiel 11-12: Reaktives Typeahead

```
// src/app/flight-typeahead/flight-typeahead.component.ts

[...]

// Hinzufügen:
import { interval } from 'rxjs';
import { switchMap, tap } from 'rxjs/operators';

[...]

@Component({
  selector: 'app-flight-typeahead',
  templateUrl: './flight-typeahead.component.html'
})
export class FlightTypeaheadComponent implements OnInit {

  // Quellen
  control = new FormControl();
  input$ = this.control.valueChanges;

  // Nicht schön. Wir reden später darüber.
  online = false;

  online$ = interval(2000).pipe(
    startWith(-1),
    map(() => Math.random() < 0.5),
    distinctUntilChanged(),
    tap(online => {
      this.online = online;
    })
  )
}
```

```
 );
[...]
}
```

Zum Simulieren des Onlinestatus kommt ein Intervall mit 2.000 Millisekunden zum Einsatz. Um sofort einen ersten Wert zu erhalten und nicht erst nach 2.000 Millisekunden, nutzt das Beispiel `startWith`. Es handelt sich dabei auch um einen `Join Operator`, der in diesem Fall den Wert `-1` am Anfang ausgibt.



Als Alternative zur Kombination von `interval` und `startWith` können Sie auch den Creation Operator `timer` verwenden:

```
online$ = timer(0, 2000).pipe( [...] );
```

Der erste Parameter gibt an, wie viel Zeit bis zur Veröffentlichung des ersten Werts verstreichen soll. Der zweite Parameter definiert die Zeitspanne zwischen den einzelnen Werten.

Der Operator `map` interessiert sich gar nicht für den aktuellen Intervallwert, sondern liefert einen zufälligen Boolean zurück. Da `Math.random` einen Zufallswert zwischen 0.0 und 1.0 liefert, ist die Chance, `true` oder `false` zu bekommen, jeweils 50%.

Mit `distinctUntilChanged` filtert das Beispiel aufeinanderfolgende Duplikate: Aus `true, true, false, false, true` wird somit `true, false, true`.

Der Operator `tap` legt schlussendlich den aktuellen Onlinestatus in einer Eigenschaft `online` ab. Das ist nicht unbedingt die feine englische Art, zumal nun sowohl `online$` als auch `online` die gleiche Information repräsentieren. Außerdem wird hiermit ein in der Welt von RxJS nicht gern gesehener Nebeneffekt eingeführt, zumal der Datenfluss andere Variablen beeinflusst. Aber keine Sorge: Diese Unschönheit werden wir im Abschnitt »Multicasting« auf Seite 279 beseitigen. Vorerst nutzen wir diese Eigenschaft auch nur zur Ausgabe des Onlinestatus im Template:

```
<!-- src/app/flight-typeahead/flight-typeahead.component.html -->
[...]
<div [ngStyle]="{'background-color': online ?
  'green' : 'red'}">Online: {{online}}</div>
```

Nun möchten wir sicherstellen, dass nur dann eine Flugsuche erfolgt, wenn der aktuelle Onlinestatus `true` ist. Genau das erreichen wir mit `combineLatest` im Konstruktor unserer `FlightSearchComponent`.

#### *Beispiel 11-13: Reaktive Flugsuche mit Onlinestatus*

```
// src/app/flight-typeahead/flight-typeahead.component.ts

// Hinzufügen:
import { combineLatest } from 'rxjs';
import { filter } from 'rxjs/operators';
```

```

import { map } from 'rxjs/operators';

[...]
constructor(private flightService: FlightService) {

    // Hinzufügen/Abändern:
    const debouncedInput$ = this.input$.pipe(
        filter(value => value.length >= 3),
        debounceTime(300),
    );

    // Hinzufügen/Abändern:
    this.flights$ = combineLatest([debouncedInput$, this.online$]).pipe(
        filter(([_, online]) => online),
        map(([input, _]) => input),
        switchMap(input => this.load(input))
    );
}

[...]

```

Das Filtern und Debouncen mit `debounceTime` lagert das Beispiel in ein eigenes Observable `debouncedInput$` aus. Danach kommt `combineLatest` zum Einsatz, um die jeweils aktuellen Werte aus `debouncedInput$` und `online$` zu kombinieren. Das Ergebnis ist ein Observable, das die beiden Werte in einem Tupel transportiert.

Der Filter interessiert sich nur für den zweiten Eintrag im Tupel – den Onlinestatus. Anstatt über den Indexwert 1 darauf zuzugreifen (`filter(tp1 => tp1[1])`), kommt eine sogenannte Destrukturierung zum Einsatz. Diese packt die ersten beiden Werte des Tupels in die Parameter `_` und `online`. Der Parametername `_` deutet einer üblichen Konvention folgend an, dass dieser Parameter hier nicht verwendet wird.

Nach dem Filtern brauchen wir nur mehr den ersten Wert aus dem Tupel – also den Suchfilter. Um keinen unnötigen Ballast herumzuschleppen, bildet `map` deswegen das Tupel auf den Suchfilter ab. Das darauffolgende `switchMap` gestaltet sich wie weiter oben beschrieben.

## combineLatest vs. withLatestFrom

Im zuvor gezeigten Beispiel lässt sich auch, abhängig von den konkreten Anforderungen, der Operator `combineLatest` durch `withLatestFrom` ersetzen (siehe Beispiel 11-14).

*Beispiel 11-14: Flugsuche mit withLatestFrom*

```

// src/app/flight-typeahead/flight-typeahead.component.ts

// Hinzufügen:
import { withLatestFrom } from 'rxjs/operators';

[...]

```

```

constructor(private flightService: FlightService) {

  const debouncedInput$ = this.input$.pipe(
    filter(value => value.length >= 3),
    debounceTime(300),
  );

  this.flights$ = debouncedInput$.pipe(
    // withLatestFrom anstatt combineLatest:
    withLatestFrom(this.online$),
    filter(([_, online]) => online),
    map(([input, _]) => input),
    switchMap(input => this.load(input))
  );
}

```

Auf den ersten Blick scheint der Operator `withLatestFrom` dasselbe wie `combineLatest` zu machen. Allerdings gibt es hier ein paar wesentliche Unterschiede. Zum einen kommt `withLatestFrom` innerhalb von `pipe` zum Einsatz und findet sich deswegen im Namensraum `rxjs/operators`. Bei `combineLatest` handelt es sich dagegen um einen Creation Operator, genau genommen um einen Join Creation Operator. Deswegen finden wir ihn im Namensraum `rxjs`.

Ein weitaus wichtigerer Unterschied besteht jedoch in der Tatsache, dass `combineLatest` tätig wird, sobald ein beliebiges der kombinierten Observables einen neuen Wert veröffentlicht. Der Operator `withLatestFrom` holt sich hingegen nur für jeden Wert des äußeren Observables den gerade aktuellen Wert des inneren.

Ein Vergleich der Marble-Diagramme der beiden Operatoren visualisieren diesen Unterschied (siehe Abbildung 11-5).

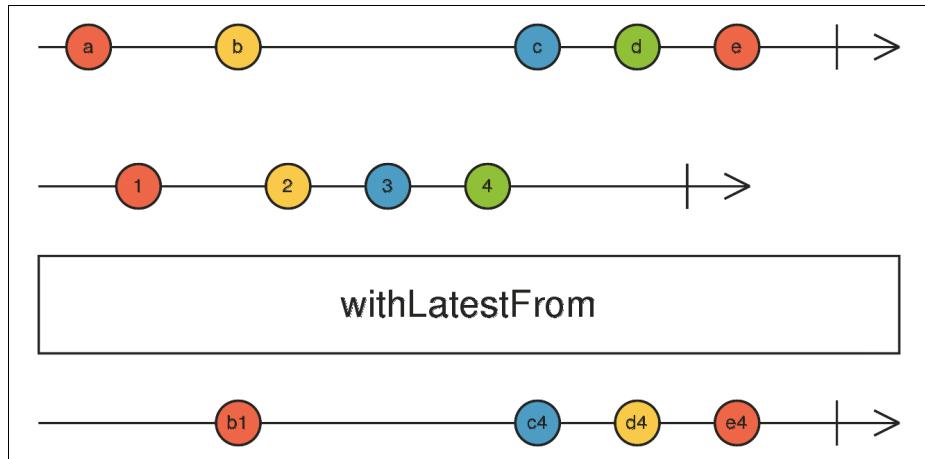


Abbildung 11-5: Marble-Diagramm für `withLatestFrom` (entnommen aus der offiziellen RxJS-Dokumentation)

Unser Beispiel sucht somit dank `withLatestFrom` nur nach Flügen, wenn eine neue Benutzereingabe erfolgt, während wir online sind. Beim Einsatz von `combineLatest` hat das Beispiel zusätzlich gesucht, nachdem der Onlinestatus auf `true` gewechselt ist.

## Der Operator `merge`

Im Gegensatz zu `combineLatest` führt `merge` die Werte aus zwei oder mehreren Observables in einem zusammen. Der Konsument des so geschaffenen Observables weiß gar nicht, aus welchen der Quell-Observables die einzelnen Werte stammen (siehe Abbildung 11-6).

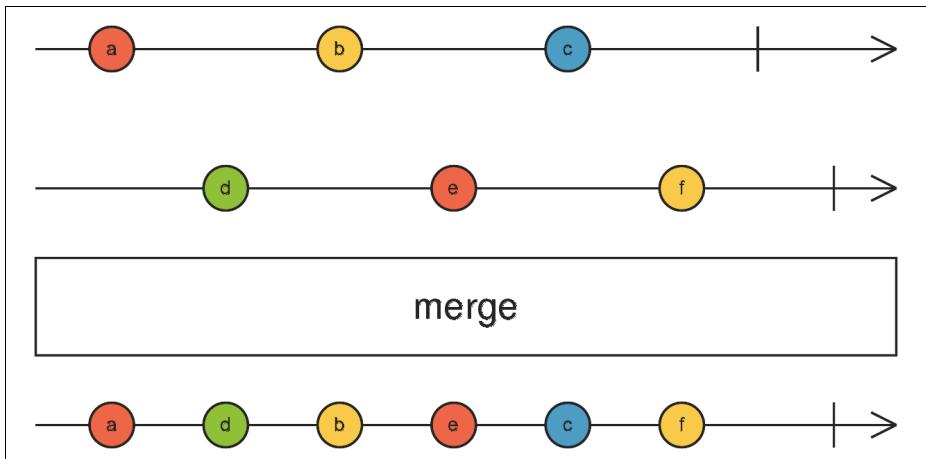


Abbildung 11-6: Marble-Diagramm für `merge` (entnommen aus der offiziellen RxJS-Dokumentation)

Zur Veranschaulichung von `merge` erweitern wir unser Beispiel um eine Schaltfläche mit der Beschriftung `Refresh`, die ein erneutes Laden der aktuellen Flüge anstößt.

Da wir hier vollständig reaktiv programmieren wollen, repräsentieren wir Klicks auf diese Schaltfläche mit einem Subject:

```
refresh$ = new Subject<Event>();
```

Dieses Subject transportiert das Eventobjekt, das den Klick repräsentiert. Das Template bindet das `click`-Event der Schaltfläche an den Aufruf der `next`-Methode dieses Subjects:

```
<button type="button" (click)="refresh$.next($event)" class="btn btn-default">
  Refresh
</button>
```

Nun wollen wir sowohl im Fall eines Klicks als auch im Fall einer Eingabe dieselbe Aktion durchführen, nämlich nach Flügen suchen. Hierfür bietet sich `merge` an (siehe Beispiel 11-15).

### Beispiel 11-15: Datenflüsse mit merge kombinieren

```
// src/app/flight-typeahead/flight-typeahead.component.ts
import { merge } from 'rxjs';

[...]

constructor(private flightService: FlightService) {

  const debouncedInput$ = this.input$.pipe(
    filter(value => value.length >= 3),
    debounceTime(300),
  );

  const inputRequest$ = combineLatest([debouncedInput$, this.online$]);

  const refreshRequest$ = this.refresh$.pipe(
    map(_ => this.control.value),
    withLatestFrom(this.online$)
  );

  this.flights$ = merge(
    inputRequest$,
    refreshRequest$,
  ).pipe(
    filter(([_, online]) => online),
    map(([input, _]) => input),
    switchMap(input => this.load(input))
  );
}

}
```

Zur Vereinfachung des Beispiels wurden zwei weitere Observables eingeführt: `inputRequest$` und `refreshRequest$`. Das Observable `inputRequest$` informiert uns, wenn aufgrund einer Benutzereingabe eine Flugsuche anzustoßen ist, und das Observable `refreshRequest$`, wenn die Flugsuche aufgrund eines Klicks auf *Refresh* erfolgen soll.

Da wir beide Observables mit `merge` kombinieren wollen, ergibt es Sinn, dass deren veröffentlichte Werte die gleiche Struktur aufweisen. In unserem Fall ist das ein Tupel mit dem Suchfilter und dem Onlinestatus. Bei `inputRequest$` ergibt sich dieser Wert aus dem oben besprochenen Einsatz von `combineLatest`.

Im Fall von `refreshRequest$` müssen wir zunächst das erhaltene Eventobjekt, das uns an dieser Stelle eigentlich gar nicht interessiert, mit `map` auf den Suchwert abbilden. Danach kombinieren wir den Suchwert mit dem aktuellen Wert aus `online$` mit `withLatestFrom`.

Anschließend können wir die beiden Observables mit `merge` zusammenführen. Egal aus welchem der Quell-Observables ein Tupel stammt: Das Beispiel führt danach die Flugsuche durch.



Auch wenn das hier verwendete Beispiel den Einsatz von `merge` plastisch veranschaulicht, möchten wir der Vollständigkeit halber darauf hinweisen, dass man es auch mit `combineLatest` umsetzen könnte:

```
this.flights$ = combineLatest([
  debouncedInput$, this.online$, this.refresh$])
  .pipe()
```

Das liegt daran, dass `combineLatest` einen neuen Wert veröffentlicht, sobald auch eines der Quell-Observables einen neuen Wert liefert. In weiterer Folge könnte man sogar den von `refresh$` gelieferten und von `combineLatest` ins Tupel aufgenommenen Wert ignorieren.

Dieser Umstand unterstreicht noch mal die eingangs aufgestellte Behauptung, dass `combineLatest` wohl der am häufigsten genutzte Join Operator ist.

## Multicasting

Auch wenn es auf den ersten Blick gar nicht so wirkt – zwischen Observable und Observer herrscht standardmäßig eine 1:1-Beziehung vor. Jeder Observer bekommt also sein eigenes Observable. Ab und an ist es jedoch nützlich, die von einem Observable veröffentlichten Werte mehreren Observern zukommen zu lassen. Bei dieser Form einer 1:N-Beziehung ist von Multicasting die Rede.

Dieser Abschnitt motiviert diesen Umstand zunächst anhand unseres Beispiels und erläutert dann die Hintergründe, die wir auch in das Beispiel einfließen lassen.

## Motivation für Multicasting

Die Notwendigkeit für Multicasting haben wir in unserem Beispiel bisher geschickt kaschiert, indem wir den aktuellen Wert von `online$` zusätzlich in der Variablen `online` speichern. Auf diese Variable können beliebig viele Stellen unserer Komponente zugreifen.

Dass das nicht schön ist, liegt auf der Hand. Lassen Sie uns also versuchen, ohne diese Variable auszukommen (siehe Beispiel 11-16).

*Beispiel 11-16: Die reaktive Flugsuche kommt nun ohne zusätzliche Variable online aus.*

```
// src/app/flight-typeahead/flight-typeahead.component.ts

[...]

export class FlightTypeaheadComponent implements OnInit {
  [...]

  // Entfernen:
  // online = false;

  online$ = interval(2000).pipe(
    startWith(-1),
```

```

    map(() => Math.random() < 0.5),
    distinctUntilChanged(),
    // Entfernen:
    // tap(online => {
    //   this.online = online;
    // })),
  );
}

[...]
}

```

Jetzt können wir im Template nicht mehr auf `online` zugreifen. Stattdessen bietet sich jedoch der Einsatz von `online$` und der `async`-Pipe an:

```
<!-- src/app/flight-typeahead/flight-typeahead.component.html -->

<div [ngStyle]="{'background-color': (online$ | async) ? 'green' : 'red'}">
  Online: {{(online$ | async)}}
</div>
```

Das scheint auf den ersten Blick auch ganz gut zu funktionieren. Das eigentliche Dilemma offenbart sich jedoch, wenn wir als Experiment diese Zeile mehrfach im Template platzieren:

```
<div [ngStyle]="{'background-color': (online$ | async) ? 'green' : 'red'}">
  Online: {{(online$ | async)}}
</div>

<div [ngStyle]="{'background-color': (online$ | async) ? 'green' : 'red'}">
  Online: {{(online$ | async)}}
</div>

<div [ngStyle]="{'background-color': (online$ | async) ? 'green' : 'red'}">
  Online: {{(online$ | async)}}
</div>

<div [ngStyle]="{'background-color': (online$ | async) ? 'green' : 'red'}">
  Online: {{(online$ | async)}}
</div>
```

Das Resultat schaut nicht allzu vertrauenswürdig aus (siehe Abbildung 11-7).



Abbildung 11-7: Verwirrung um den Onlinestatus (unterschiedliche Hintergrundfarben wurden zur besseren Lesbarkeit entfernt)

Auch wenn Ihre Ausgabe von der hier gezeigten ein wenig abweichen kann, werden Sie zur selben Zeit unterschiedliche Werte von `online$` erhalten. Das liegt an der erwähnten 1:1-Beziehung, die standardmäßig zwischen Observable und Observer vorherrscht. Somit starten Sie pro `async-Pipe` ein eigenes Intervall, das wiederum zu eigenen Zufallswerten führt (siehe Beispiel 11-16).

Man sagt auch, dass hier ein kaltes Observable (*Cold Observable*) vorliegt. Was das genau bedeutet, erläutert der nächste Abschnitt.

## Hot vs. Cold Observables

In Hinblick auf das Multicasting-Verhalten unterscheidet man zwischen zwei Arten von Observables: Cold und Hot Observables.

*Cold Observables* haben die folgenden Eigenschaften:

- Pro Observer existiert ein eigenes Observable.
- Die Ausführung startet erst, wenn sich ein Observer anmeldet (man sagt, es sei »lazy«).

Dem stehen die *Hot Observables* mit den folgenden Eigenschaften gegenüber:

- Ein Hot Observable kann mehrere Observer bedienen. Alle erhalten Zugriff auf die veröffentlichten Werte.
- Die Ausführung startet unabhängig von der Anzahl an verbundenen Observer.

Die meisten Observables, mit denen wir zu tun haben, sind cold. Eine Ausnahme stellen Subjects dar – diese sind immer hot. Sogenannte *Multicasting Operators* machen aus einem Cold Observable ein Hot Observable. Die wohl am häufigsten verwendeten Vertreter dieser Gruppe sind `share` und `shareReplay`.

Während `share` das Observable lediglich in ein Hot Observable umwandelt, löst `shareReplay` auch das Late-Subscriber-Problem, indem es die letzten n Werte zwischenspeichert. Neue Observer erhalten somit sofort diese Werte, um sich ein Bild vom aktuellen Zustand zu machen.



Technisch gesehen, nutzen `share` und `shareReplay` unter der Motorhaube ein Subject. Im ersten Fall handelt es sich um ein normales Subject und im zweiten Fall um ein ReplaySubject.

Um `online$` mit `shareReplay` zum Hot Observable zu machen, reicht die folgende Ergänzung:

```
// src/app/flight-typeahead/flight-typeahead.component.ts

// Hinzufügen:
import { shareReplay } from 'rxjs/operators';

[...]
```

```

online$ = interval(2000).pipe(
  startWith(-1),
  map(() => Math.random() < 0.5),
  distinctUntilChanged(),

  // Hinzufügen:
  shareReplay({ refCount: true, bufferSize: 1 })

  // Alternative mit share
  // share()
);

```

In der Regel werden Sie genau die hier gezeigte Überladung mit den hier verwendeten Werten nutzen:

#### *refCount*

Der Wert `true` gibt an, dass das Quell-Observable (hier das Intervall) geschlossen wird, sobald sich der letzte Observer abmeldet.

#### *bufferSize*

Gibt an, wie viele Werte zur Lösung des Late-Subscriber-Problems zwischen gespeichert werden sollen.

Sowohl `share` als auch `shareReplay` starten das Quell-Observable erst, sobald die erste Anmeldung vorliegt.

## Fehlerbehandlung

Sobald im Rahmen des Datenflusses ein Fehler auftritt, veröffentlichen Observable diesen und schließen sich. Das bedeutet, dass der Observer nur noch den Fehler empfängt. Danach erhält er jedoch keine weiteren Werte mehr. Dieses Verhalten ist durchaus so gewollt, zumal es zu verhindern gilt, dass Anwendungen basierend auf einem Fehlerzustand eventuell weitere fehlerhafte Aktionen durchführen.

Möchten Sie das verhindern, muss Ihr Observable den Fehler behandeln. Das kann zum Beispiel mit dem Operator `catchError` erfolgen (siehe Beispiel 11-17).

#### *Beispiel 11-17: Fehlerbehandlung mit catchError*

```

// src/app/flight-typeahead/flight-typeahead.component.ts

// Hinzufügen:
import { catchError } from 'rxjs/operators';
import { of } from 'rxjs';

[...]

this.flights$ = merge(
  refreshRequest$,
  inputRequest$,
).pipe(
  filter(([_, online]) => online),
  map(([input, _]) => input),
  catchError(error => {
    if (error === 'no connection') {
      return of([]);
    }
    return throwError(error);
  })
);

```

```

switchMap(input => this.load(input).pipe(
    // Fehler behandeln:
    catchError(err => {
        console.error('err', err);
        return of([]);
    })
)),
// Hier ist es zu spät!
// switchMap hat bereits geschlossen
// catchError(err => {
//     console.error('err', err);
//     return of([]);
// })
);

```

Bitte beachten Sie, dass Sie Fehler in dem Observable behandeln müssen, in dem sie auch auftreten. Im gezeigten Fall handelt es sich um das innerhalb von `switchMap` verwendete Observable, das bei einem Serverfehler oder einer fehlenden Internetverbindung scheitert.

Der Operator `catchError` nimmt den aktuellen Fehler entgegen und liefert ein »Ersatz-Observable« zurück, auf das es umschaltet. Das Beispiel erzeugt ein Observable mit einer leeren Ergebnismenge mittels `of`.



Um den Fehler anderweitig reaktiv weiterzuverarbeiten, können Sie ihn über ein Subject veröffentlichen:

```

error$ = new BehaviorSubject<unknown>(null);
[...]

catchError(err => {
    error$.next(error.message);
    return of([]);
})
[...]

```

Außerdem können Sie den Fehler nach dem Protokollieren auch weiterfließen lassen, was dann aber zu einem Schließen des Observables führt:

```

import { throwError } from 'rxjs';

[...]

catchError(err => {
    error$.next(error);
    return throwError(err);
})

```

Das entspricht einem `catch` mit einem darin platzierten `throw` in der Welt der imperativen Programmierung.

Um diese Fehlerbehandlungslogik auszuprobieren, können Sie in den Entwicklerwerkzeugen Ihres Browsers eine Offlinesituation simulieren (siehe Abbildung 11-8).

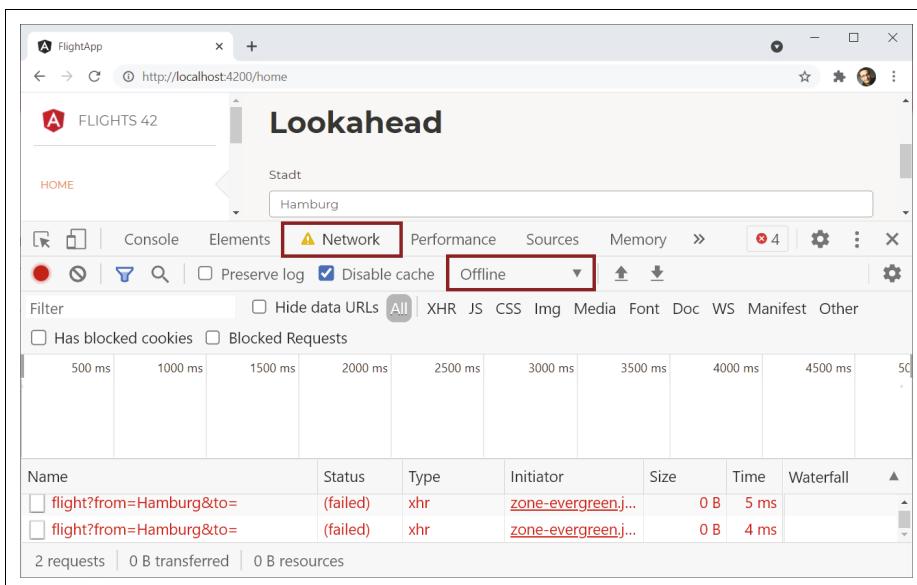


Abbildung 11-8: Offlinesituation im Browser simulieren

Da wir den Fehler innerhalb von `switchMap` behandeln müssen, kommt es hier leider zu einer Verschachtelung und somit zu schlecht lesbarem Code. Eine einfache Lösung besteht darin, die Fehlerbehandlung in einer Hilfsfunktion bzw. -methode, an die `switchMap` delegiert, aufzurufen. In unserem Fall bietet sich dafür die Methode `load` an (siehe Beispiel 11-18).

#### Beispiel 11-18: Fehlerbehandlung in Hilfsmethode

```
// src/app/flight-typeahead/flight-typeahead.component.ts
```

```
[...]
```

```
@Component({
  selector: 'app-flight-typeahead',
  templateUrl: './flight-typeahead.component.html'
})
export class FlightTypeaheadComponent implements OnInit {

  [...]

  constructor(private flightService: FlightService) {
    [...]
    // Hier ist alles wie immer.

    this.flights$ = merge(
      refreshRequest$,
      inputRequest$,
```

```

        ).pipe(
            filter(([_, online]) => online),
            map(([input, _]) => input),
            switchMap(input => this.load(input)),
        );
    }

    load(from: string): Observable<Flight[]> {
        console.log('from', from);

        return this.flightService.find(from, '').pipe(
            // Fehlerbehandlung innerhalb von load:
            catchError(err => {
                console.error('err', err);
                return of([]);
            })
        );
    }
}

```

## Observables schließen

Um Memory-Leaks zu verhindern, sollten Sie Ihre Observer wieder abmelden, sobald Sie sie nicht mehr benötigen. Das gilt insbesondere beim Einsatz von Observables, die sich nicht selber schließen und Ressourcen wie Event-Handler oder Timer bzw. Intervalle binden. Beispiele dafür sind die vom HttpClient gelieferten Observables. Sie schließen sich nach Versenden der Antwortnachricht selbst. Auch der Angular-Router schließt beim Routenwechsel bereitgestellte Observables, z.B. jene, die aktuelle URL-Parameter repräsentieren.

Für das Abmelden von Observern haben Sie die folgenden Möglichkeiten:

### *Explizites Abmelden*

Das explizite Abmelden erfolgt wie oben gezeigt durch Aufruf der Methode `unsubscribe`. Hier besteht die Gefahr darin, dass darauf vergessen wird. Eventuell gibt es auch bestimmte Situationen, in denen der Programmfluss den Aufruf von `unsubscribe` nicht erreicht.

### *Implizites Abmelden*

Zum impliziten Schließen nutzen Sie Operatoren wie z.B. `take`, die den Observer unter bestimmten Umständen automatisch abmelden. Der Einsatz von `$obs.take(2).pipe(...).subscribe(...)` meldet den Observer zum Beispiel nach zwei empfangenen Werten ab. Etwas universeller einsetzbar ist der Operator `takeUntil`. Er nimmt ein weiteres Observable entgegen. Dieses Observable informiert den Operator, wenn es an der Zeit ist, den Observer abzumelden.

### *Einsatz der async-Pipe*

Auch die `async`-Pipe schließt Observer implizit, sobald sie nicht mehr benötigt werden. Im Gegensatz zur zuvor beschriebenen Option erfolgt ihr Einsatz jedoch nicht programmatisch, sondern deklarativ.

Generell empfiehlt es sich, sofern irgendwie möglich, die Datenbindung mit der `async`-Pipe durchzuführen. Diese Strategie ist einfach, und die Pipe kümmert sich ohne weiteres Zutun um das Abmelden. Außerdem lässt sich damit auch die Datenbindung beschleunigen (siehe Kapitel 13).

Ist der Einsatz von `async` nicht möglich, sollten Sie dem impliziten Schließen mit Operatoren wie `take` oder `takeUntil` den Vorzug geben. Hier sieht man auf den ersten Blick, wie lange das Observable leben wird. Außerdem läuft man nicht Gefahr, später auf `unsubscribe` zu vergessen. Ein Beispiel, das den Einsatz von `takeUntil` zeigt, findet sich in Beispiel 11-19.

*Beispiel 11-19: Observable mit takeUntil schließen*

```
// src/app/flight-typeahead/flight-typeahead.component.ts

// Hinzufügen:
import { OnDestroy } from '@angular/core';
import { takeUntil } from 'rxjs/operators';

[...]

@Component({
  selector: 'app-flight-typeahead',
  templateUrl: './flight-typeahead.component.html'
})
// OnDestroy zu implementieren hinzufügen:
export class FlightTypeaheadComponent implements OnInit, OnDestroy {

  // Hinzufügen:
  close$ = new Subject<void>();

  flights$: Observable<Flight[]>;
  constructor(private flightService: FlightService) {
    this.flights$ = [...];
    this.flights$  

      // Hinzufügen:  

      .pipe(takeUntil(this.close$))  

      .subscribe(f => console.debug('flights', f));
  }
  // Hinzufügen:
  ngOnDestroy(): void {
    this.close$.next();
  }
  [...]
}
```

Das Beispiel übergibt das neue Subject `close$` an `takeUntil` und versendet einen neuen (leeren) Wert in `ngOnDestroy`. Werden mehrere Observables via `takeUntil` mit `close$` verbunden, kann die Komponente all diese mit einem Schlag durch den Aufruf der `next`-Methode schließen.

## Reaktive Services

Um den Fokus auf das Wesentliche zu lenken, haben sich die Beispiele in diesem Kapitel bis jetzt auf die `FlightTypeaheadComponent` beschränkt. Zur Sicherstellung der Wartbarkeit und Testbarkeit sollten Sie jedoch einige Teile in eigene Services auslagern. Deswegen wollen wir hier zum Abschluss ein kleines Refactoring vorschlagen.

Der Onlinestatus soll künftig von einem eigenen Service angeboten werden, zumal diese Information sicher auch in anderen Komponenten nützlich ist und in automatisierten Tests (siehe Kapitel 12) simuliert werden muss. Diesen Service nennen wir `OnlineService`. Sie können ihn wie gehabt mit der Angular CLI erzeugen:

```
ng generate service flight-typeahead/online
```

Seine Implementierung findet sich in Beispiel 11-20.

*Beispiel 11-20: Service zur Verwaltung des Onlinestatus*

```
// src/app/flight-typeahead/online.service.ts

import { Injectable } from '@angular/core';
import { interval } from 'rxjs';
import { distinctUntilChanged, map, shareReplay, startWith } from 'rxjs/operators';

@Injectable({
  providedIn: 'root'
})
export class OnlineService {
  readonly online$ = interval(2000).pipe(
    startWith(0),
    map(_ => Math.random() < 0.5),
    distinctUntilChanged(),
    shareReplay({ bufferSize: 1, refCount: true })
);
}
```

Etwas komplexer ist der neu einzurichtende `FlightTypeaheadService`, der die Logik von unserer Komponente erhält. Auch er lässt sich mit der CLI generieren:

```
ng generate service flight-typeahead/flight-typeahead
```

Die Implementierung dieses Service erhält für alle Datenquellen ein Observable bzw. ein Subject (siehe Beispiel 11-21).

*Beispiel 11-21: Der FlightTypeaheadService erhält die Logik für die reaktive Flugsuche.*

```
// src/app/flight-typeahead/flight-typeahead.service.ts

import { Injectable } from '@angular/core';
import { BehaviorSubject, combineLatest, merge, Observable, of, Subject } from 'rxjs';
import { catchError, debounceTime, filter, map, switchMap, withLatestFrom }
    from 'rxjs/operators';
import { Flight } from '../flight-booking/flight';
import { FlightService } from '../flight-booking/flight.service';
import { OnlineService } from './online.service';

@Injectable({
  providedIn: 'root'
})
export class FlightTypeaheadService {

  // Datenquellen
  private fromSubject = new BehaviorSubject<string>('');
  private refreshSubject = new Subject<void>();
  readonly online$ = this.onlineService.online$;

  // Datensenken
  readonly flights$: Observable<Flight[]>;
  errorSubject = new Subject<unknown>();
  readonly error$ = this.errorSubject.asObservable();

  constructor(
    private flightService: FlightService,
    private onlineService: OnlineService,
  ) {

    const debouncedInput$ = this.fromSubject.pipe(
      filter(input => input.length >= 3),
      debounceTime(300)
    );

    const inputRequest$ = combineLatest([debouncedInput$, this.online$]);

    const refreshRequest$ = this.refreshSubject.pipe(
      switchMap(_ => this.fromSubject),
      withLatestFrom(this.online$)
    );

    this.flights$ = merge(
      inputRequest$,
      refreshRequest$,
    ).pipe(
      filter(([_, online]) => online),
      map(([input, _]) => input),
      switchMap(input => this.load(input))
    );
  }

}
```

```

    search(from: string): void {
      this.fromSubject.next(from);
    }

    refresh(): void {
      this.refreshSubject.next();
    }

    clearError(): void {
      this.errorSubject.next(null);
    }

    private load(from: string) {
      return this.flightService.find(from, '').pipe(
        catchError(err => {
          this.errorSubject.next(err);
          return of([]);
        })
      );
    }
  }
}

```

Interessant ist hier, dass sämtliche Subjects private und sämtliche Observables readonly (sowie implizit public) sind. Dabei handelt es sich um eine gängige Praxis. Private Subjects verhindern, dass andere Systembestandteile für Verwirrung sorgen, indem sie Werte über unsere Subjects veröffentlichen. Das readonly bei den öffentlichen Observables verhindert hingegen, dass andere Systembestandteile die Referenzen auf unsere Observables überschreiben.

Außerdem fällt auf, dass die öffentlichen Methoden keinen Rückgabewert (void) haben. Auch das ist üblich. Mögliche Zustandsänderungen kommuniziert der Service ohnehin über die veröffentlichten Observables.

Durch das Auslagern der Logik in einen eigenen Service gestaltet sich die Implementierung der eigentlichen FlightTypeaheadComponent sehr geradlinig (siehe Beispiel 11-22).

*Beispiel 11-22: Die FlightTypeaheadComponent ist nun mit dem FlightTypeaheadService verdrahtet.*

```

import { Component, OnInit } from '@angular/core';
import { FormControl } from '@angular/forms';
import { FlightTypeaheadService } from './flight-typeahead.service';

@Component({
  selector: 'app-flight-typeahead',
  templateUrl: './flight-typeahead.component.html'
})
export class FlightTypeaheadComponent implements OnInit {

  // Verdrahten:
  flights$ = this.service.flights$;
}

```

```

online$ = this.service.online$;
error$ = this.service.error$;

control = new FormControl();

constructor(private service: FlightTypeaheadService) {
}

ngOnInit(): void {
    // An Service delegieren:
    this.control.valueChanges.subscribe(from => {
        this.service.search(from);
    });
}

// Hinzufügen:
refresh(): void {
    this.service.refresh();
}

// Hinzufügen:
clearError(): void {
    this.service.clearError();
}
}

}

```

Im Wesentlichen muss die Komponente nur mehr Observables zum Lesen vom Service entgegennehmen und zum richtigen Zeitpunkt seine Methoden aufrufen.

Der Vollständigkeit halber zeigt Beispiel 11-23 das Template unserer nun aufgeräumten FlightTypeaheadComponent.

*Beispiel 11-23: Template der aktualisierten FlightTypeaheadComponent*

```

<h2 class="title">Typeahead</h2>

<!-- Hinzufügen: -->
<div *ngIf="error$ | async as error">
    {{error}}
    [<a (click)="clearError()">Close!</a>]
</div>

<div class="control-group">
    <label>Stadt</label>
    <input [formControl]="control" class="form-control">
    <!-- Aktualisieren: -->
    <button (click)="refresh()" class="btn btn-default">Refresh</button>
</div>

<div [ngStyle]="{{'background-color': (online$ | async)? 'green' : 'red'}}">
    Online: {{online$ | async}}
</div>

```

```

<table class="table table-striped">
  <tr *ngFor="let f of flights$ | async">
    <td>{{f.id}}</td>
    <td>{{f.from}}</td>
    <td>{{f.to}}</td>
    <td>{{f.date | date:'dd.MM.yyyy HH:mm'}}</td>
  </tr>
</table>

```

Neu sind hier eigentlich nur die Datenbindung für die Schaltfläche *Refresh*, die sich jetzt direkt an die neue `refresh`-Methode bindet, sowie die Ausgabe von eventuellen Fehlern, die die Komponente via `error$` kommuniziert.

Bei Letzterem kommt ein in der Angular-Welt üblicher Kunstkniff zum Einsatz: Die Direktive `*ngIf` fragt den Wert von `error$` mit `async` ab und schreibt ihn mittels `as` in die Template-Variable `error`. Letzteres verhindert, dass das Template diesen Wert erneut für die Ausgabe mit `async` abrufen muss.

## Zusammenfassung

RxJS erlaubt es, asynchrone Datenflüsse mit linearen Codestrecken darzustellen. Das macht den Code leichter lesbar und vereinfacht somit auch die Wartung. Operatoren helfen beim Anpassen der Datenflüsse. Sie erlauben es zum Beispiel, verschiedene Datenflüsse zusammenzuführen oder auch Daten zu verändern.

Daneben existieren Operatoren für die Fehlerbehandlung sowie für das Multicasting. Letzteres bedeutet, dass ein Observable seine Werte mit mehreren Empfängern, den sogenannten Observern, teilt. Zum Verhindern verschachtelter Observables kommen Flattening Operators zum Einsatz. Sie transportieren die Werte eines inneren Observables auf das äußere.

Um Memory-Leaks zu verhindern, sollten Observables nach ihrer Verwendung geschlossen werden. Das kann entweder explizit durch Aufruf von `unsubscribe` oder implizit durch den Einsatz von Operatoren wie `take` oder `takeUntil` erfolgen. Wann immer möglich, sollte die `async`-Pipe zum Einsatz kommen. Sie ermöglicht zum einen die performantere OnPush-Strategie für die Datenbindung und kümmert sich zum anderen um das Schließen von Observables.



# Testautomatisierung

Durch die Nutzung von Frameworks wie Angular und ihrer Konzepte wandert immer mehr Logik in das Frontend. Manuelles Testen reicht nach kurzer Zeit nicht mehr aus, um die fortlaufende Stabilität einer Applikation zu gewährleisten.

Glücklicherweise kommt Angular mit einigen Konzepten, die bei der Automatisierung von Tests helfen. In diesem Kapitel beschäftigen wir uns damit, indem wir Tests für unsere Demoanwendung entwickeln.

## Jasmine und Karma

Die Angular CLI inkludiert standardmäßig das Testing-Framework Jasmine (<https://jasmine.github.io>). Um Jasmine-Tests auszuführen, kommt der Test-Runner Karma (<https://karma-runner.github.io>) zum Einsatz. Damit startet die CLI einen oder mehrere konfigurierte Browser und führt darin die Jasmine-Tests aus.

## Aufbau eines Jasmine-Tests

Jasmine gibt uns Funktionen zum Definieren von Testfällen und Testsuites. Während Testfälle Teile unserer Lösung auf Funktionstüchtigkeit prüfen, gruppieren Testsuites zusammengehörige Testfälle (siehe Beispiel 12-1).

*Beispiel 12-1: Beispielhafter Jasmine-Test*

```
// src/app/basics.spec.ts

// Prüfling
const add = (a: number, b: number) => a + b;

// Testsuite
describe('Add', () => {

    // vor jedem Testfall ausführen
    beforeEach(() => {
        console.log('Vorbereitungsaufgaben ...');
    });
})
```

```

// Testfall
it('correctly adds 1 and 2', () => {
    // Arrange
    const a = 1;
    const b = 2;

    // Act
    const c = add(a, b);

    // Assert
    expect(c).toBe(3);
});

});

```

Tests sind per definitionem in Dateien mit der Endung `.spec.ts` zu platzieren. Diese Endung unterstreicht auch die Tatsache, dass Tests als Spezifikationen für das gewünschte Verhalten angesehen werden können.

Die Funktion `describe` richtet eine Testsuite ein und `it` einen Testfall. Diesen haben wir im Beispiel nach dem AAA-Muster aufgebaut: Die *Arrange*-Phase bereitet die Testausführung vor, die *Act*-Phase führt die zu testende Aktion aus, und die *Assert*-Phase prüft, ob das gewünschte Ergebnis erreicht wurde.

Falls Sie die gleichen Vorbereitungsaufgaben für jeden Test einer Suite benötigen, können Sie diese auch in einen `beforeEach`-Block auslagern. Analog dazu lässt sich mit `afterEach` eine Funktion festlegen, die die CLI nach jedem Testfall anstößt. Daneben führen `beforeAll` und `afterAll` die festgelegten Routinen einmal vor allen bzw. nach allen Testfällen der Suite aus.

In der *Assert*-Phase kommen in der Regel `expect`-Aufrufe zum Einsatz. Diese Funktion lässt sich mit zahlreichen Methoden, die den jeweiligen Wert prüfen, verketten. Das nachfolgende Beispiel zeigt ein paar ausgewählte Möglichkeiten:

```

expect(c).not.toBe(4);
expect(c).not.toBeNull();
expect(c).toBeGreaterThanOrEqual(0);

const str = `Result: ${c}`;
expect(str).toContain('Result');
expect(str).toMatch(/Result/);

```

Diese Methoden nennen sich auch *Matcher*. Am besten verwenden Sie die Codevollständigung Ihrer IDE, um sich über die angebotenen *Matcher* zu informieren.

Zur Vereinfachung beinhaltet das hier gezeigte Beispiel ebenfalls den zu testenden Building-Block – in der deutschsprachigen Literatur auch als *Prüfling* bekannt.

Testsuites lassen sich übrigens beliebig ineinanderschachteln, indem man innerhalb eines `describe`-Blocks nochmals `describe` aufruft. Der Idee von *Behavior Driven Development* (BDD) folgend, ist es auch üblich, dass die Namen der Suites und Testfälle die getesteten Anforderungen widerspiegeln. In unserem Beispiel lautet diese auf *Add correctly adds 1 and 2*.

## Tests mit Karma ausführen

Um Jasmine-Tests innerhalb einer oder mehrerer JavaScript-Laufzeitumgebungen auszuführen, verwendet die CLI Karma. Als Laufzeitumgebung nutzt Karma in der Regel die auf dem System installierten Browser. Standardmäßig konfiguriert die CLI Karma für die Nutzung von Chrome. Dazu platziert sie die folgende Zeile in der `karma.conf.js` im Projekthauptverzeichnis:

```
browsers: ['Chrome'],
```

Daneben lassen sich auch andere Browser wie Firefox, Opera oder Safari als Ausführungsumgebung nutzen. Details dazu finden Sie in der Dokumentation von Karma (<http://karma-runner.github.io>).

Bevor wir unsere oben gezeigte Testsuite ausführen, ändern wir noch das `describe` in `fdescribe` ab:

```
fdescribe('Add', () => {
  [...]
});
```

Dabei handelt es sich um einen Kunstknaif: Existieren mit `fdescribe` definierte Testsuites, werden nur diese ausgeführt. Jene, die `describe` verwenden, lässt Karma in diesem Fall außen vor.

Damit lassen sich sämtliche Testsuites, die die CLI für die bisher erzeugten Building-Blocks generiert hat, deaktivieren. Das ist auch notwendig, zumal wir sie nicht gepflegt haben und deswegen die meisten davon scheitern werden. In einem richtigen Projekt würde man diese Testfälle nachziehen. Hier wollen wir uns allerdings zum Lernen auf ein paar wenige beschränken. Das lässt sich erreichen, indem Sie jene Testfälle, die es auszuführen gilt, mit `fit` anstatt mit `it` definieren.

Im Übrigen werden Sie gleich bei der Testausführung ein paar Kompilierungsfehler auf der Konsole erhalten. Diese ergeben sich auch, weil wir bis jetzt die von der CLI generierten Testfälle nicht gewartet haben. Um sich damit nicht belasten zu müssen, können Sie die entsprechenden Zeilen aus diesen Testfällen oder die gesamten betroffenen Testfälle löschen. Bitte beachten Sie, dass Sie diese Fehler beheben müssen, damit die restlichen Testfälle überhaupt ausgeführt werden.

Um Karma jetzt zu starten, nutzen Sie die folgende Anweisung:

```
ng test
```

Karma startet nun Chrome und führt darin unsere Tests aus (siehe Abbildung 12-1).

Die Ausführung beschränkt sich hier auf den Testfall *Add correctly adds 1 and 2*. Die restlichen von der CLI generierten Testfälle werden zwar auch erkannt, aber aufgrund der Nutzung von `fdescribe` nicht angestoßen. Die Information im oberen Bereich weist darauf hin, und die Namen dieser ignorierten Testfälle erscheinen ausgegraut.

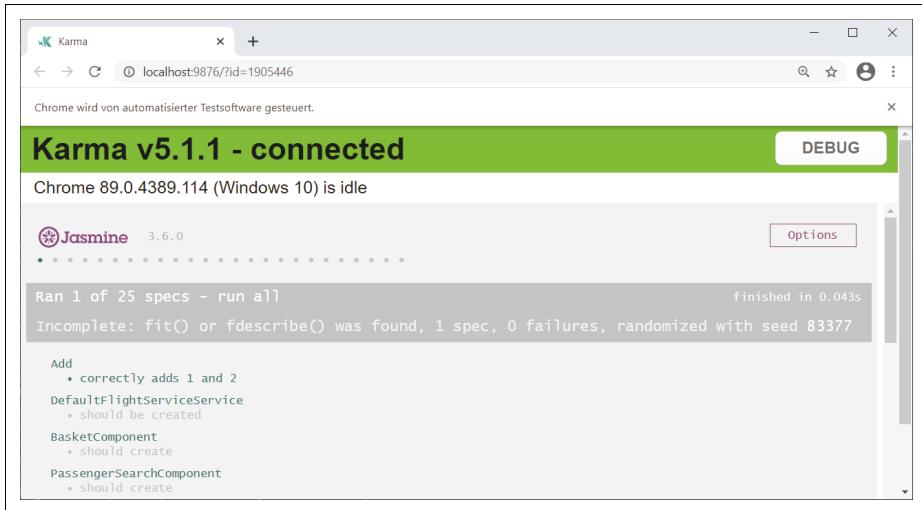


Abbildung 12-1: Karma führt in Chrome einen Testfall aus.

## Karma auf dem Build-Server

Das Starten von Anwendungen mit grafischen Benutzeroberflächen ist auf dem Build-Server beim automatisierten Bauen und Testen hinderlich. Deswegen bietet sich der Einsatz eines Browsers wie Headless Chrome an. Dabei handelt es sich um eine Variante von Chrome, die keine UI aufweist.

Der folgende Eintrag in der `karma.conf.js` aktiviert Headless Chrome:

```
browsers: ['ChromeHeadless'],
```

Daneben ist Karma anzuweisen, die Testergebnisse zu protokollieren. Der Build-Server kann diese Protokolle importieren und basierend darauf über den Zustand des Systems informieren. Als Format für solche Protokolle bietet sich das XML-basierte JUnit-Format an. Dabei handelt es sich um einen De-facto-Standard in der Welt von Java, den die meisten Build-Server unterstützen.

Um zu steuern, auf welche Weise Karma Testergebnisse protokolliert, kommen sogenannte *Reporter* zum Einsatz. Der Reporter für JUnit lässt sich über `npm` installieren:

```
npm i karma-junit-reporter -D
```

Das Paket ist in der `karma.conf.js` als Plug-in zu registrieren (siehe Beispiel 12-2).

### Beispiel 12-2: Karma-Konfiguration mit JUnit-Reporter

```
// karma.conf.js
```

[...]

```
module.exports = function (config) {
```

```

config.set({
  ...
  plugins: [
    ...
    // Hinzufügen:
    require('karma-junit-reporter'),
  ],
  ...
  // junit hinzufügen:
  reporters: ['progress', 'kjhtml', 'junit'],
  // Konfiguration für JUnit-Reporter hinzufügen:
  junitReporter: {
    outputDir: 'test-reports'
  },
  ...
  browsers: ['ChromeHeadless'],
  ...
});
});

```

Außerdem ist der Reporter `junit` unter `reporters` einzutragen und eine Konfiguration dafür zu ergänzen. Die anderen beiden standardmäßig registrierten Reporter informieren auf der Konsole über den Testfortschritt (`progress`) und geben im geöffneten Browserfenster weitere Informationen über die Testausführung aus (`kjhtml`, steht für `karma-jasmine-html-reporter`).

Wenn der Build-Prozess die Tests nun mit `ng test` ausführt, schreibt Karma die XML-basierte JUnit-Datei in das konfigurierte Verzeichnis.

## Angular und Jasmine

Da die meisten Angular-Building-Blocks lediglich Klassen sind, könnten wir bereits mit den bis jetzt diskutierten Aspekten Angular-Lösungen testen. Allerdings bringt Angular zur Vereinfachung noch ein paar Konstrukte mit sich, die den Brückenschlag zu Jasmine vereinfachen. Diese betrachten wir hier etwas näher.

### Komponenten mit dem TestBed testen

Eines der zentralen Konstrukte, die Angular zum Testen bereitstellt, ist das `TestBed`. Wie bei einem Prüfstand in der Elektrotechnik oder im Maschinenbau lassen sich dort die zu testenden Komponenten »anbringen« und mit Signalen versorgen. Reagieren die Komponenten wie erwartet, haben sie den jeweiligen Test bestanden.

Ein `TestBed` besteht aus einem Angular-Modul, dem sogenannten *Testing Module*, bei dem sich die Prüflinge und deren Abhängigkeiten registrieren lassen. Außerdem kann das `TestBed` Komponenten instanziieren. Dabei werden abhängige Services injiziert, und auch das Template wird zum Leben erweckt. Letzteres erlaubt das Interagieren mit Komponenten auf der DOM-Ebene.

Um das Testen von Komponenten mit dem TestBed zu veranschaulichen, erweitern wir den Test, den die CLI für die FlightSearchComponent generiert hat (siehe Beispiel 12-3).

*Beispiel 12-3: Test für die FlightSearchComponent*

```
// src/app/flight-booking/flight-search/flight-search.component.spec.ts

import { HttpClientModule } from '@angular/common/http';
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { FlightSearchComponent } from './flight-search.component';

// Hinzufügen:
import { SharedModule } from 'src/app/shared/shared.module';
import { FlightCardComponent } from '../flight-card/flight-card.component';

fdescribe('FlightSearchComponent', () => {
  let component: FlightSearchComponent;
  let fixture: ComponentFixture<FlightSearchComponent>;

  beforeEach(async () => {
    // Prüfling und seine Abhängigkeiten hinterlegen:
    await TestBed.configureTestingModule({
      imports: [ HttpClientModule, SharedModule ],
      declarations: [ FlightSearchComponent, FlightCardComponent ]
    })
    .compileComponents();
  });

  beforeEach(() => {
    // Komponente erzeugen
    fixture = TestBed.createComponent(FlightSearchComponent);
    // Komponenteninstanz abrufen
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  // Anpassen:
  it('should have no loaded flights initially', () => {
    expect(component.flights.length).toBe(0);
  });
});
```

Der erste beforeEach-Block konfiguriert das Modul des TestBed. Es erhält mit der FlightSearchComponent den Prüfling. Außerdem erhält es weitere Module und Komponenten, von denen FlightSearchComponent abhängig ist.

Die Methode compileComponents kompiliert die Templates der registrierten Komponenten. Da diese Methode asynchron ist, kommt im betrachteten beforeEach-Block eine Kombination aus async und await zum Einsatz.



Die hier betrachtete Konstellation testet die FlightSearchComponent gemeinsam mit der FlightCardComponent. Um solche komponentenübergreifenden Tests zu vermeiden, könnten Sie statt der FlightCard Component eine Dummy-Komponente mit dem Selektor von Flight CardComponent registrieren. Hierbei ist auch von flachen (*shallow*)

Tests die Rede. Diese haben den Vorteil, dass sie simpler sind, weil sie sich auf einen einzigen Prüfling beschränken. Sie lassen sich dadurch schneller ausführen, und Fehler können eindeutig dem Prüfling zugeordnet werden.

Der zweite `beforeEach`-Block erzeugt die `FlightSearchComponent` und erhält ein *Fixture* (deutsch etwa Teststellung) vom Typ `ComponentFixture<FlightSearchComponent>`. Dieses Fixture repräsentiert die verschiedenen Aspekte unserer Komponente. Es weist unter anderem die folgenden Member auf:

#### *componentInstance*

Liefert das Objekt, das die Komponente repräsentiert. Im hier betrachteten Beispiel ist diese Komponenteninstanz ein Objekt vom Typ `FlightBookingComponent`.

#### *nativeElement*

Liefert das DOM-Objekt der Komponente.

#### *debugElement*

Das Debug-Element erleichtert den Zugriff auf weitere Aspekte des Prüflings. Es erlaubt das Abfragen von untergeordneten DOM-Knoten oder dessen Services.

#### *detectChanges*

Anders als bei der Ausführung einer Angular-Anwendung triggern Tests die Änderungsverfolgung nicht automatisch. Wollen wir die Datenbindungen im Template des Prüflings aktualisieren, müssen wir diese Methode aufrufen.

Der unterhalb der beiden `beforeEach`-Blöcke dargestellte Testfall nutzt die Komponenteninstanz, um zu prüfen, ob anfangs bereits Flüge dargestellt werden.

Zum Ausführen dieses Tests können Sie wieder `ng test` verwenden. Bitte beachten Sie, dass wir auch hier zu Demonstrationszwecken `fdescribe` statt `describe` nutzen, damit die betrachtete Testsuite zu jener kleinen Menge gehört, die wir ausführen wollen.

## Arbeiten mit Attrappen (Mocks)

Dependency Injection erlaubt das Austauschen von Abhängigkeiten. Gerade bei automatisierten Tests ist das häufig notwendig. Um den Grund dafür zu veranschaulichen, erweitern wir hier die `FlightSearchComponent`, sodass sie eine Suche mit leeren Suchparametern verweigert (siehe Beispiel 12-4).

*Beispiel 12-4: Die FlightSearchComponent prüft nun, ob Suchkriterien vorliegen.*

```
// src/app/flight-search/flight-search.component.ts
```

```
[...]
```

```
export class FlightSearchComponent implements OnInit {
```

```
[...]
```

```

search(): void {
    // Hinzufügen:
    if (!this.from || !this.to) {
        return;
    }
    [...]
}
[...]
}

```

Lassen Sie uns auch die im letzten Abschnitt betrachtete Testsuite um zwei – zugegeben naive – Tests für die aktualisierte search-Methode ergänzen (siehe Beispiel 12-5).

*Beispiel 12-5: Naiver Versuch, die Methode search zu testen*

```

// src/app/flight-booking/flight-search/flight-search.component.spec.ts
[...]
fdescribe('FlightSearchComponent', () => {
    [...]
    it('should search for flights when from and to are given', () => {
        component.from = 'Hamburg';
        component.to = 'Graz';
        component.search();

        expect(component.flights.length).toBe(3);
    });

    it('should *not* search for flights *without* from and to', () => {
        component.from = '';
        component.to = '';
        component.search();

        expect(component.flights.length).toBe(0);
    });
})

```

Dieser Versuch, die Methode search zu testen, ist aus folgenden Gründen naiv:

- Wir können nicht davon ausgehen, dass es immer genau drei Flüge von Hamburg nach Graz gibt.
- Die Methode search ist asynchron. Deswegen können wir nicht unmittelbar nach ihrer Ausführung auf die geladenen Flüge zugreifen.
- Dieser Test testet die gesamte Aufrufkette bis hin zum Backend. Das macht ihn langsam und fehleranfällig.
- Durch das Testen der gesamten Aufrufkette lassen sich eventuelle Fehler nicht eindeutig dem Prüfling zuordnen.

Glücklicherweise lassen sich alle diese Probleme mit einem einzigen Konzept lösen: Mocking. Dabei werden Abhängigkeiten durch Attrappen ersetzt. Im betrachteten Fall tauschen wir den `DefaultFlightService` gegen den `DummyFlightService` aus. Dieser liefert synchron immer drei fixe Flüge, ohne an weitere Systembestandteile zu delegieren.

Beachten Sie, dass wir hier nicht testen, ob der Backend-Aufruf funktioniert. Wir testen lediglich, ob die Methode `search` korrekt an den `FlightService` delegiert bzw. das Delegieren verweigert, wenn keine Suchparameter vorliegen.



Tests, die die gesamte Aufrufkette testen, nennt man auch *End-2-End-Tests*. Testen wir hingegen nur einen einzelnen Systembestandteil, z.B. eine Komponente oder einen Service, ist von *Unit-Tests* die Rede. Sie liefern rasch Feedback und können somit häufig ausgeführt werden. Da sie von ihren tatsächlichen Abhängigkeiten isoliert sind, zeigen sie sich auch weniger fehleranfällig.

Unter anderem deswegen ist man sich mittlerweile darüber einig, dass die meisten Tests Unit-Tests sein sollten. Um zu testen, ob einzelne Systembestandteile auch korrekt zusammenspielen, braucht es jedoch auch ein paar wenige End-2-End-Tests.

Hierzu kommen häufig Technologien zum Einsatz, die Browser fernsteuern und so Benutzerinteraktionen simulieren. In der Vergangenheit hat die Angular CLI mit *Protractor* eine solche Lösung geliefert. Allerdings wurde Protractor mit Angular CLI 12 abgekündigt und wird auch nicht durch einen offiziellen Nachfolger ersetzt.

Zum Glück finden sich am Markt einige kommerzielle sowie freie Alternativen. Einer Umfrage des Angular-Teams zufolge nutzten bereits vor der Abkündigung von Protractor über 60% der Angular-Entwickler das Werkzeug *Cypress* (<https://www.cypress.io>).

Die hier verwendete Implementierung des `DummyFlightService` entspricht jener aus Kapitel 5 (siehe Beispiel 12-6).

#### Beispiel 12-6: `DummyFlightService`

```
// src/app/dummy-flight.service.ts

[...]

@Injectable({
  providedIn: 'root'
})
export class DummyFlightService implements FlightService {

  constructor() { }

  find(from: string, to: string): Observable<Flight[]> {
    return of([
      { id: 1, from: 'Frankfurt', to: 'Flagranti', date: '2022-01-02T19:00+01:00' },
      { id: 2, from: 'Frankfurt', to: 'Kognito', date: '2022-01-02T19:30+01:00' },
      { id: 3, from: 'Frankfurt', to: 'Mallorca', date: '2022-01-02T20:00+01:00' }
    ]);
  }
}
```

Um den DefaultFlightService für unseren Test durch diese Attrappe zu ersetzen, müssen wir lediglich im TestBed einen entsprechenden Provider einrichten (siehe Beispiel 12-7).

*Beispiel 12-7: Das TestBed nutzt nun den DummyFlightService.*

```
// src/app/flight-booking/flight-search/flight-search.component.spec.ts  
[...]  
  
// Hinzufügen:  
import { FlightService } from '../flight.service';  
import { DummyFlightService } from '../dummy-flight.service';  
  
fdescribe('FlightSearchComponent', () => {  
    [...]  
  
    beforeEach(async () => {  
        await TestBed.configureTestingModule({  
            imports: [ HttpClientModule, SharedModule ],  
            declarations: [ FlightSearchComponent, FlightCardComponent ],  
  
            // Hinzufügen:  
            providers: [  
                { provide: FlightService, useClass: DummyFlightService }  
            ]  
        })  
        .compileComponents();  
    });  
  
    [...]  
});
```

Um zu verhindern, dass Beispiele aus anderen Kapiteln uns hier einen Strich durch die Rechnung machen, sollten Sie an dieser Stelle auch sicherstellen, dass Ihre FlightSearchComponent den Typ FlightService als Token verwendet und keine Provider auf Komponentenebene einrichtet (siehe Beispiel 12-8).

*Beispiel 12-8: Die FlightSearchComponent nutzt den Typ FlightService als Token.*

```
// src/app/flight-search/flight-search.component.ts  
  
import { Component, OnInit } from '@angular/core';  
import { Flight } from '../flight';  
import { FlightService } from '../flight.service';  
  
@Component({  
    selector: 'app-flight-search',  
    templateUrl: './flight-search.component.html',  
    styleUrls: ['./flight-search.component.scss']  
})  
export class FlightSearchComponent implements OnInit {
```

```

from = 'Hamburg';
to = 'Graz';
flights: Array<Flight> = [];
selectedFlight: Flight | null = null;
delayFilter = false;

basket: { [key: number]: boolean } = {
  3: true,
  5: true
};

constructor(private flightService: FlightService) {}

ngOnInit(): void {}

search(): void {

  if (!this.from || !this.to) {
    return;
  }

  this.flightService.find(this.from, this.to).subscribe({
    next: (flights) => {
      this.flights = flights;
    },
    error: (err) => {
      console.debug('Error', err);
    }
  });
}

select(f: Flight): void {
  this.selectedFlight = f;
}
}

```

Führen Sie nun die Tests mit `ng test` aus, sollten die beiden neuen Tests erfolgreich sein.

Provider, die Sie – wie gerade gezeigt – im TestBed einrichten, tauschen übrigens nur globale Services aus. Für Services, die auf Komponentenebene eingerichtet wurden, ist hingegen eine andere Vorgehensweise notwendig (siehe Beispiel 12-9).

#### *Beispiel 12-9: Service auf Komponentenebene austauschen*

```
// src/app/flight-booking/flight-search/flight-search.component.spec.ts
[...]

beforeEach(async () => {
  // Prüfling und seine Abhängigkeiten hinterlegen:
  await TestBed.configureTestingModule({

```

```

imports: [ HttpClientModule, SharedModule ],
declarations: [ FlightSearchComponent, FlightCardComponent ],
providers: [
  { provide: FlightService, useClass: DummyFlightService}
]
})
.compileComponents();

// Service auf Komponentenebene tauschen:
 TestBed.overrideComponent(FlightSearchComponent, {
  set: {
    providers: [
      { provide: FlightService, useClass: DummyFlightService}
    ]
  }
});
});

[...]

```

Die Methode `overrideComponent` erlaubt das Überschreiben sämtlicher mit dem Component-Dekorator festgelegter Metadaten. Dazu zählen auch Provider für lokale Services.

## Gray-Box-Tests mit Spys

Die bisher betrachteten Tests waren allesamt sogenannte *Black-Box-Tests*. Das bedeutet, dass uns die interne Arbeitsweise der Prüflinge nicht interessiert hat, so lange sie das gewünschte Verhalten an den Tag gelegt haben. *Gray-Box-Tests* treffen hingegen ein paar Annahmen über interne Vorgänge. Im Fall der `FlightSearch Component` könnte solch ein Test zum Beispiel prüfen, ob er die `find`-Methode des `FlightService` mit den korrekten Parametern aufruft.

Für diese Aufgabe bietet Jasmine sogenannte *Spys*. Es handelt sich dabei um Proxies, die einzelne Funktionen oder Methoden dekorieren und sich merken, mit welchen Parametern sie aufgerufen wurden. Danach können Spys an die eigentliche Funktion bzw. Methode delegieren oder stattdessen eine Mock-Implementierung ausführen.

Um einen Spy für die Methode `find` des verwendeten `FlightService` zu erzeugen, übergeben wir die Serviceinstanz an die Funktion `spyOn` (siehe Beispiel 12-10).

### Beispiel 12-10: Gray-Box-Testing mit Spys

```

// src/app/flight-booking/flight-search/flight-search.component.spec.ts
[...]

fdescribe('FlightSearchComponent', () => {
  let component: FlightSearchComponent;
  let fixture: ComponentFixture<FlightSearchComponent>;

```

```

// Hinzufügen:
let flightService: FlightService;

[...]

beforeEach(() => {
  fixture = TestBed.createComponent(FlightSearchComponent);
  component = fixture.componentInstance;
  fixture.detectChanges();

  // Hinzufügen:
  flightService = fixture.debugElement.injector.get(FlightService);
  spyOn(flightService, 'find').and.callThrough();
    // spyOn ist global und muss deswegen nicht importiert werden!
});

[...]

it('should search for flights when from and to are given', () => {
  component.from = 'Hamburg';
  component.to = 'Graz';
  component.search();

  expect(component.flights.length).toBe(3);

  // Hinzufügen:
  expect(flightService.find).toHaveBeenCalled();
});

it('should *not* search for flights *without* from and to', () => {
  component.from = '';
  component.to = '';
  component.search();

  expect(component.flights.length).toBe(0);

  // Hinzufügen:
  expect(flightService.find).not.toHaveBeenCalled();
});

});

```

Als zweiten Parameter nimmt `spyOn` den Namen der zu dekorierenden Methode als String entgegen. Der Service selbst lässt sich über den Injector der jeweiligen Komponente beziehen. Dieser findet sich in ihrem Debug-Element.

Die mit `spyOn` verketteten Methoden legen das Verhalten des Spys fest. Der Aufruf von `and.callThrough()` führt zum Beispiel dazu, dass der Spy den Aufruf nicht nur protokolliert, sondern danach auch die dekorierte Methode anstößt. Alternativ dazu könnte der Spy auch einen fixen Wert zurückliefern, an eine Mock-Implementierung delegieren oder eine Exception auslösen:

```

import { of } from 'rxjs';
[...]

```

```

// Einen fixen Wert zurückliefern.
spyOn(flightService, 'find').and.returnValue(of([
{id: 1, from: 'A', to: 'B', date: ''}]));
// Eine Fake-Implementierung (Mock) aufrufen.
spyOn(flightService, 'find').and.callFake((from, to) => of([
{id: 1, from: 'A', to: 'B', date: ''}]));
// Einen Fehler werfen.
spyOn(flightService, 'find').and.throwError(new Error('Manfred needs some coffee!'));

```

Dank des eingeführten Spys können die Testfälle nun prüfen, ob die Methode tatsächlich angestoßen wurde. Dazu kommt der Matcher `toHaveBeenCalled` zum Einsatz. Alternativ dazu lässt sich auch prüfen, ob die Methode mit bestimmten Parametern aufgerufen wurde bzw. wie häufig sich der Prüfling an die Methode gewendet hat:

```

expect(flightService.find).toHaveBeenCalledWith('Hamburg', 'Graz');
expect(flightService.find).toHaveBeenCalledTimes(1);

```

## HTTP-Zugriffe simulieren

Auch beim Testen von Services wie dem `DefaultFlightService` ist es wünschenswert, konkrete HTTP-Zugriffe zu simulieren. Da der `DefaultFlightService` jedoch direkt den `HttpClient` nutzt, ist dieser auszutauschen. Dabei hilft Angular mit seinem `HttpClientTestingModule`.

Kommt das `HttpClientTestingModule` anstelle des `HttpClientModule` zum Einsatz, führt der `HttpClient` keine HTTP-Zugriffe durch. Stattdessen protokolliert er lediglich die angeforderten Aufrufe in Form von Request-Objekten. Ihr Test kann nun diese Request-Objekte abrufen und selbst mit Fake-Ergebnissen beantworten.

Um diesen Mechanismus zu nutzen, importieren Sie das `HttpClientTestingModule` anstelle des `HttpClientModule` in das `TestBed` (siehe Beispiel 12-11).

### *Beispiel 12-11: Mocken des HttpClient*

```

// src/app/flight-booking/default-flight.service.spec.ts

import { TestBed } from '@angular/core/testing';
import { HttpClientTestingModule, HttpTestingController } from '@angular/common/http/testing';
import { DefaultFlightService } from './default-flight.service';

fdescribe('DefaultFlightService', () => {
  let service: DefaultFlightService;
  let httpCtrl: HttpTestingController;

  beforeEach(() => {
    TestBed.configureTestingModule({
      // HttpClientTestingModule (!) importieren:
      imports: [HttpClientTestingModule],

```

```

    // Kann beim Einsatz von Tree-Shakable Providers entfallen:
    providers: [DefaultFlightService]
  });

  service = TestBed.inject(DefaultFlightService);

  // Hinzufügen:
  httpCtrl = TestBed.inject(HttpTestingController);
});

it('should call API with parameters', () => {

  // Methode in Prüfling anstoßen.
  // Wichtig: Erst subscribe führt die (Fake-)HTTP-Anfrage aus!
  service.find('Hamburg', 'Graz').subscribe(flights => {
    // Ergebnis prüfen.
    expect(flights.length).toBe(1);
    expect(flights[0].from).toBe('A');
  });

  // Prüfen, ob eine Anfrage für die erwartete URL vorliegt.
  const req = httpCtrl.expectOne(
    'http://demo.ANGULARarchitects.io/api/flight?from=Hamburg&to=Graz');

  // Fake-Antwort bereitstellen.
  req.flush([{"id": 1, "from": "A", "to": "B", "date": ''}]);

  // Jetzt geht die Ausführung im subscribe-Block oben weiter!
});
});

```

An dieser Stelle versagt der Auto-Import von Visual Studio Code sehr häufig. Deswegen ist es gut, zu wissen, dass sich das `HttpClientTestingModule` im Namensraum `@angular/common/http/testing` befindet.

Im selben Namensraum befindet sich auch der `HttpTestingController`. Damit lassen sich anstehende Anfragen in Erfahrung bringen. Der Testfall ruft mit dem `DefaultFlightService` Flüge ab. Die Methode `find` delegiert an den `HttpClient`, woraufhin dieser die gewünschte HTTP-Anfrage protokolliert.

Das zurückgelieferte Observable veröffentlicht vorerst keinen Wert. Deswegen wird auch der Code innerhalb von `subscribe` noch nicht ausgeführt.

Mit `expectOne` prüft der `HttpTestingController`, ob der erwartete Aufruf vorliegt, und ermittelt das dazu passende Request-Objekt. Danach beantwortet der Test die Anfrage selbst, woraufhin der an `subscribe` übergebene Handler angestoßen wird.

Der vorliegende Test prüft somit, ob sich der `DefaultFlightService` an die richtige URL wendet und ob er das vom `HttpClient` erhaltene Ergebnis korrekt verarbeitet.

Als Alternative zur Methode `expectOne` können Sie übrigens mit `match` die Menge der protokollierten Abfragen etwas flexibler filtern:

```

const requests = httpCtrl.match(r =>
  r.url.includes('/api/flight')
  && r.method === 'GET'
  && r.params.get('from') === 'Hamburg');

const req = requests[0];

```

## Asynchrone Tests

Bis jetzt konnten wir dank Mocking sämtliche Tests synchron gestalten. Selbst wenn diese mit Observables hantieren, wurden die Fake-Daten stets ohne Verzögerung geliefert.

In manchen Fällen ist das jedoch nicht möglich. Das hat zur Folge, dass Jasmine nicht wissen kann, ob die Testausführung bereits abgeschlossen ist oder ob der Test lediglich auf weitere asynchrone Daten wartet.

Zur Demonstration testen wir den `FlightTypeaheadService` aus Kapitel 11 (siehe Beispiel 12-12).

*Beispiel 12-12: Testsuite für den FlightTypeaheadService*

```
// src/app/flight-typeahead/flight-typeahead.service.spec.ts

@Injectable()
class DummyOnlineService {
  online$ = new BehaviorSubject<boolean>(true);
}

fdescribe('FlightTypeaheadService', () => {
  let service: FlightTypeaheadService;

  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [
        { provide: OnlineService, useClass: DummyOnlineService },
        { provide: FlightService, useClass: DummyFlightService },
      ]
    });
  });

  service = TestBed.inject(FlightTypeaheadService);
});

[...]
});
```

Sowohl der `OnlineService`, der nach dem Zufallsprinzip meldet, ob der Browser online ist, als auch der `FlightService` werden durch einen Mock ersetzt. Der dazu verwendete `DummyOnlineService` findet sich direkt in der Datei mit dem Test. Beim `DummyFlightService` handelt es sich um jenen aus Kapitel 5.

Der eigentliche Testfall muss Jasmine wie erwähnt mitteilen, wann er fertig ist. Dazu nutzt er eine `done`-Funktion, die er als Argument entgegennimmt (siehe Beispiel 12-13).

*Beispiel 12-13: Asynchroner Test mit done-Funktion*

```
// src/app/flight-typeahead/flight-typeahead.service.spec.ts  
[...]  
  
it('should load flights (done)', (done: DoneFn) => {  
  
  service.flights$.subscribe(flights => {  
    expect(flights.length).toBe(3);  
    expect(flights[0].from).toBe('Wien');  
    done();  
  });  
  
  service.search('Wien');  
  
});  
[...]
```

Bitte beachten Sie auch, dass zuerst mit subscribe ein Observer mit den Expectations registriert wird und erst danach die Suche angestoßen werden kann.

Um die Arbeit mit solchen asynchronen Tests ein wenig zu vereinfachen, bietet Angular die Hilfsfunktion waitForAsync. Sie ummantelt den Testfall und informiert Jasmine automatisch, sobald alle asynchronen Aufgaben abgearbeitet wurden (siehe Beispiel 12-14).

*Beispiel 12-14: Asynchroner Test mit waitForAsync*

```
// src/app/flight-typeahead/flight-typeahead.service.spec.ts  
  
import { waitForAsync } from '@angular/core/testing';  
  
[...]  
  
it('should load flights', waitForAsync(() => {  
  
  service.flights$.subscribe(flights => {  
    expect(flights.length).toBe(3);  
    expect(flights[0].from).toBe('Wien');  
  });  
  
  service.search('Wien');  
  
}));
```

Somit kommen wir hier ohne done-Funktion aus.

Etwas schwieriger wird es, wenn wir auch das Debouncing von 300 Millisekunden (debounceTime(300)) in unserer reaktiven Flugsuche testen wollen. Hier gilt es nämlich, das Verstreichen bestimmter Zeiteinheiten zu simulieren. Für solche Aufgaben bietet Angular die Funktion fakeAsync an, mit der ebenfalls der Testfall zu ummanteln ist (siehe Beispiel 12-15).

### Beispiel 12-15: Debouncing mit fakeAsync testen

```
// src/app/flight-typeahead/flight-typeahead.service.spec.ts

import { fakeAsync, tick } from '@angular/core/testing';
[...]
it('should not debounce', fakeAsync(() => {

  let counter = 0;
  service.flights$.subscribe(flights => {
    counter++;
    expect(flights.length).toBe(3);

    if (counter === 1) {
      expect(flights[0].from).toBe('Graz');
    }

    if (counter === 2) {
      expect(flights[0].from).toBe('Wien');
    }
  });

  service.search('Graz');
  tick(400);
  service.search('Wien');
  tick(400);
}));
```

Die Funktion `tick` simuliert hier das Verstreichen von 400 Millisekunden.



Technisch gesehen, führt die Funktion `tick` alle anstehenden Macro- und Micro-Tasks aus. Macro-Tasks sind unter JavaScript z.B. Timer (`timeout`, `interval`) oder anstehende Event-Handler. Micro-Tasks sind z.B. Promises. Möchten Sie nur alle anstehenden Micro-Tasks abarbeiten, können Sie auch die Funktion `flushMicrotasks` nutzen.

## Templates mit DOM-Zugriffen testen

Tests können auch direkt mit dem Template interagieren. Im Fall unserer `FlightSearchComponent` könnten Sie zum Beispiel prüfen, ob die Schaltfläche `Search` ausgegraut (`disabled`) ist.

Zur Vereinfachung gehen wir im Folgenden davon aus, dass die Eigenschaft `disabled` der Schaltfläche `Search` an den Ausdruck `!from || !to` gebunden ist:

```
<!-- src/app/flight-search/flight-search.component.html -->
[...]
<button class="btn btn-default" (click)="search()" [disabled]="!from || !to">
  Search
</button>
```

Der hier verwendete asynchrone Testfall wartet zunächst mit `whenStable`, bis der initiale Aufbau des Suchformulars abgeschlossen sind (siehe Beispiel 12-16).

*Beispiel 12-16: Testfall mit Zugriff auf Template*

```
// src/app/flight-booking/flight-search/flight-search.component.spec.ts

[...]
import { By } from '@angular/platform-browser';

fdescribe('FlightSearchComponent', () => {
  let component: FlightSearchComponent;
  let fixture: ComponentFixture<FlightSearchComponent>;
  let flightService: FlightService;

  [...]

  it('should have a disabled search button w/o params', async () => {
    await fixture.whenStable();

    // get from
    const from = fixture
      .debugElement
      .query(By.css('input[name=from]'))
      .nativeElement;

    from.value = '';
    from.dispatchEvent(new Event('input'));
    fixture.detectChanges();

    // get to
    const to = fixture
      .debugElement
      .query(By.css('input[name=to]'))
      .nativeElement;

    to.value = '';
    to.dispatchEvent(new Event('input'));
    fixture.detectChanges();

    // get disabled
    const disabled = fixture
      .debugElement
      .query(By.css('button'))
      .properties.disabled;

    expect(disabled).toBeTruthy();
  });
});
```

Über das Debug-Element ermittelt der Test jene DOM-Elemente, die die beiden Eingabefelder `from` und `to` repräsentieren. Dazu nutzt er die Methode `query` gemeinsam mit `By`, um diese Elemente anhand von CSS-Selektoren zu identifizieren.

Wichtig ist an dieser Stelle, dass nach dem Setzen der Werte sowohl ein `input`-Event als auch die Change Detection manuell ausgelöst werden müssen. Ersteres

ist notwendig, damit `@angular/forms` die Wertänderung erkennt, und Letzteres ist notwendig, damit Angular die geänderten Werte über die Datenbindung zurück-schreibt.



Leider existiert keine vollständig automatische Datenbindung für Tests. Es gibt zwar Mechanismen, die den Aufruf von `detectChanges` mancherorts unnötig machen, da diese Mechanismen jedoch nicht in jeder Konstellation funktionieren, empfehlen wir, `detectChanges` manuell aufzurufen, um Verwirrung zu vermeiden.

Auf die gleiche Weise bezieht der Test das DOM-Element, das die Schaltfläche wi-derspiegelt. Den Wert von `disabled` findet er in der Auflistung `properties`.

## Direktiven testen

Da Direktiven in der Regel – direkt oder indirekt – das DOM verändern, lassen sie sich auch mit der im letzten Abschnitt gezeigten Vorgehensweise testen. Allerdings brauchen Sie eine Komponente, die die Direktive verwendet. Dabei kann es sich um eine Dummy-Komponente handeln, die zu Testzwecken die Direktive in ihrem Template aufruft.

Wollten wir zum Beispiel testen, ob die Validierungsdirektive `required` dazu führt, dass Angular nach dem Leeren des Eingabefelds eine Fehlermeldung mit der Klasse `error-required` einblendet, könnten wir das wie folgt prüfen:

```
// Anzahl an Elementen mit Klasse error-required ermitteln.  
const count = fixture  
    .debugElement  
    .queryAll(By.css('.error-required'))  
    .length;  
  
expect(count).toBe(1);
```

Die Klasse `error-required` könnte im Template wie folgt zum Einsatz kommen:

```
<div class="error error-required" *ngIf="f?.controls?.from?.hasError('required')">  
  Dieses Feld ist ein Pflichtfeld.  
</div>
```

## Pipes testen

Um eine Pipe zu testen, reicht es in der Regel aus, sie selbst mit `new` zu instanziieren und ihre `transform`-Methode aufzurufen. Verwendet die Pipe Services, können Sie auch die Pipe als Service deklarieren und als solchen mit `TestBed.inject` samt allen Abhängigkeiten abrufen (siehe Beispiel 12-17).

*Beispiel 12-17: Eine Pipe testen*

```
// src/app/shared/city.pipe.spec.ts  
  
import { TestBed } from '@angular/core/testing';
```

```

import { CityPipe } from './city.pipe';
import { CityService } from './city.service';

fdescribe('CityPipe', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [CityPipe, CityService]
    });
  });

  it('should transform Hamburg to HAM', () => {
    const pipe = TestBed.inject(CityPipe);
    const result = pipe.transform('Hamburg', 'short');
    expect(result).toBe('HAM');
  });
});

```

## Testabdeckung ermitteln

Die Angular CLI kann Sie beim Finden von Kandidaten für weitere Testfälle unterstützen, indem sie ermittelt, welche Codestrecken wie gut durch bestehende Tests abgedeckt sind. Hierbei spricht man auch von der Testabdeckung.

Sie müssen dazu lediglich `ng test` mit dem Schalter `--code-coverage` aufrufen:

```
ng test --code-coverage
```

Daraufhin führt die CLI Ihre Tests aus und hinterlegt im Ordner `coverage/projekt-name` (z. B. `coverage/flight-app`) eine statische Website, die über die Testabdeckung informiert (siehe Abbildung 12-2).

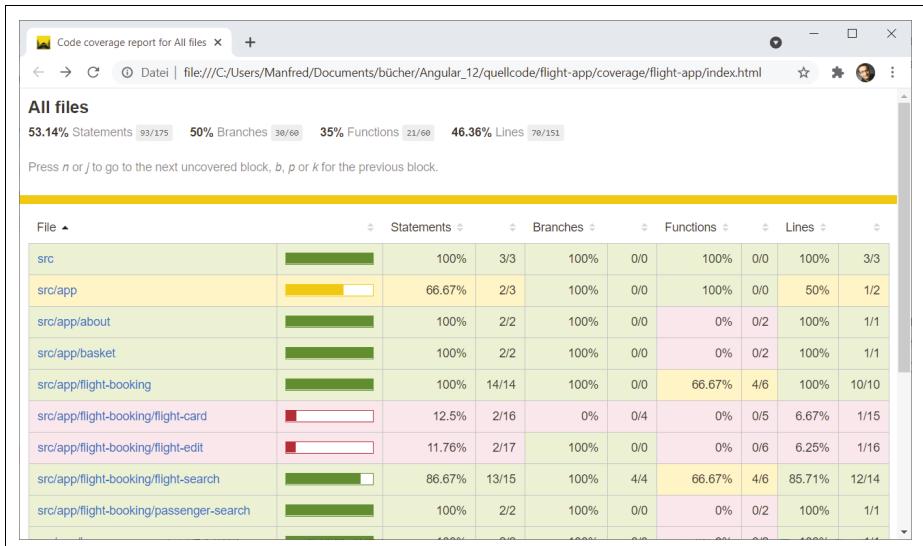


Abbildung 12-2: Generierte Website mit Informationen zur Testabdeckung

Wählen Sie eine der angezeigten Dateien aus, erfahren Sie sogar, welche Zeilen wie häufig von Ihren Tests durchlaufen wurden (siehe Abbildung 12-3).

```
32
33 2x      if (!this.from || !this.to) {
34 1x          return;
35      }
36
37 1x      this.flightService.find(this.from, this.to).subscribe({
38          next: (flights) => {
39              this.flights = flights;
40          },
41          error: (err) => {
42              console.debug('Error', err);
43          }
44      );
45
46  }
47
48  select(f: Flight): void {
49      this.selectedFlight = f;
50  }
51
52  }
53
```

Abbildung 12-3: Testabdeckung auf Dateiebene

## Zusammenfassung

Automatisierte Tests geben während der Entwicklung Feedback und helfen bei der Sicherstellung der gewünschten Codequalität. Die Angular CLI richtet ohne Ihr Zutun die nötige Infrastruktur dafür ein. Diese beinhaltet die Ausführungsumgebung Karma, die Ihre Tests in einem oder mehreren Browsern ausführt und die Testergebnisse protokolliert.

Die Tests sind mit Jasmine zu schreiben. Angular bietet einige Helferlein, um die Arbeit mit Jasmine zu vereinfachen. Dazu gehört das TestBed, mit dem Sie Building-Blocks für den Test registrieren bzw. austauschen können. Außerdem bietet Angular Unterstützung für asynchrone Tests. Um zu sehen, welche Bereiche Ihrer Anwendung Potenzial für weitere Tests bieten, können Sie mit der CLI die Testabdeckung ermitteln und grafisch darstellen.

# Performancetuning

Performance war eines der Architekturziele von Angular. Deswegen verwundert es auch nicht, dass Angular von Haus aus sehr schnell ist. Außerdem bringt das Framework ein paar Stellschrauben mit, um die Performance bei großen Anwendungen weiter zu optimieren.

Dieses Kapitel geht auf zwei Optionen ein: *OnPush* verbessert die Performance der Datenbindung, und *Lazy Loading* erlaubt es, einzelne Anwendungsteile erst bei Bedarf zu laden.

## Optimierte Datenbindung mit OnPush

Die Datenbindung in Angular ist sehr schnell. Der Angular-Compiler erzeugt beispielsweise für Datenbindungen Code, der sich von der JavaScript-Engine im Browser besonders gut optimieren lässt. Solche sogenannten monomorphen Codestrecken erreichen dank Laufzeitoptimierungen fast die Performance nativer Anwendungen.

Angular nutzt jedoch standardmäßig Dirty-Checking. Das bedeutet, dass jeweils nach dem Ausführen von Event-Handlern alle Datenbindungen auf Änderungen geprüft werden. Dieser Umstand macht die Entwicklung mit Angular einfacher, zumal sich so jedes beliebige Objekt binden lässt. Auf der anderen Seite kostet die ständige Überprüfung sämtlicher Komponenten auch CPU-Zyklen.

Genau hier setzt die Datenbindungsstrategie OnPush an. In diesem Modus findet Angular zielgerichtet heraus, welche Bindungen zu aktualisieren sind. Damit diese Strategie funktioniert, muss die Anwendung besondere Datenstrukturen verwenden, nämlich *Immutables* und *Observables*.

Dieser Abschnitt geht auf *Immutables* und *Observables* ein und zeigt, wie sich diese Konstrukte gemeinsam mit OnPush einsetzen lassen.

## Datenbindung visualisieren

Bevor wir uns OnPush zuwenden, möchten wir mit einem Experiment die Änderungsverfolgung von Angular visualisieren. Die Idee ist, unsere Flugkarten aufblitzen zu lassen, wenn Angular sie auf Änderungen überprüft. Hierfür ändern wir die Hintergrundfarbe für eine halbe Sekunde.

Dazu erhält zunächst die FlightCardComponent die folgenden Objekte injiziert:

```
// src/app/flight-card/flight-card.component.ts  
[...]  
import { ElementRef, NgZone } from '@angular/core';  
[...]  
  
constructor(private elm: ElementRef, private ngZone: NgZone) {  
    [...]  
}
```

Die ElementRef repräsentiert den DOM-Knoten der FlightCard. NgZone repräsentiert *Zone.js*. Dabei handelt es sich um jenen Mechanismus, der Angular Bescheid gibt, wenn Event-Handler gelaufen sind.

Mit diesen beiden Objekten lässt sich die in Beispiel 13-1 gezeigte blink-Methode in der FlightCard einrichten.

### Beispiel 13-1: Änderungsverfolgung visualisieren

```
// src/app/flight-card/flight-card.component.ts  
  
blink(): void {  
    // Unorthodox code for visualizing Change Detection  
    // don't use it in production ...  
    this.elm.nativeElement.firstChild.style = 'background-color: crimson';  
  
    this.ngZone.runOutsideAngular(() => {  
        setTimeout(() => {  
            this.elm.nativeElement.firstChild.style = '';  
        }, 500);  
    });  
}
```

Dieser Code muss ohne Datenbindung auskommen, da ansonsten die Visualisierung weitere Datenbindungen und somit auch weitere Visualisierungen nach sich zöge. Aus dem gleichen Grund umgeht *blink* auch *Zone.js* mit *runOutsideAngular*.

In weiterer Folge benötigt unser Experiment eine Methode, die lediglich einen Flug ändert. Wir nennen sie *delay* und richten sie in der FlightSearchComponent ein.

### Beispiel 13-2: Flug abändern

```
// src/app/flight-search/flight-search.component.ts  
[...]  
  
delay(): void {
```

```

// From ISO-String to Date object
const date = new Date(this.flights[0].date);

// Add 15 Minutes (1000 * 60 * 15 msec)
date.setTime(date.getTime() + 1000 * 60 * 15);

// Mutate original ISO-String
this.flights[0].date = date.toISOString();
}

```

Das Template der FlightSearchComponent erhält eine Schaltfläche für delay. Diese können wir nach der Schaltfläche *Search* einfügen:

```

<!-- src/app/flight-search/flight-search.component.html -->
[...]

<button [...]>
  Search
</button>

<button
  *ngIf="flights.length > 0"
  class="btn btn-default"
  (click)="delay()"
  Delay 1st Flight
</button>

```

Wenn Sie nun Ihre Anwendung starten und nach Flügen suchen, sollten Sie sehen, dass Angular sämtliche Flüge im Anschluss an alle Events prüft. Das gilt zum Beispiel für das blur-Event, das der Browser auslöst, wenn Sie ein Suchfeld verlassen.

Aber auch das Verzögern des ersten gefundenen Flugs mittels delay veranlasst Angular, sämtliche Flüge zu prüfen. Genau dieses Verhalten lässt sich mit OnPush optimieren.

## Immutables

Der Name lässt es schon vermuten: Immutables sind Datenstrukturen, die nicht veränderbar sind. Ändern sich die damit beschriebenen Objekte, müssen Sie das gesamte Immutable durch ein neues austauschen. Um Änderungen zu entdecken, müssen Frameworks somit nur prüfen, ob das Immutable als Ganzes getauscht wurde, anstatt sich über seine einzelnen Eigenschaften auf dem Laufenden zu halten. Oder um es etwas technischer auszudrücken: Das Framework muss nur prüfen, ob sich die Objektreferenz geändert hat.

Zur Veranschaulichung zeigt Beispiel 13-3 eine Variante der zuvor eingeführten delay-Methode, die auf Immutables basiert:

*Beispiel 13-3: Delay unter Nutzung von Immutables*

```
// src/app/flight-search/flight-search.component.ts
[...]
```

```

delay(): void {
  const oldFlights = this.flights;
  const oldFlight = oldFlights[0];
  const oldFlightDate = new Date(oldFlight.date);

  const newFlightDate = new Date(oldFlightDate.getTime() + 1000 * 60 * 15);

  const newFlight = {
    ...oldFlight, date: newFlightDate.toISOString()
  };

  const newFlights = [
    newFlight, ...oldFlights.slice(1)
  ];

  this.flights = newFlights;
}

```

Zugegeben, diesen Code könnte man auch kürzer schreiben. Allerdings wollten wir zur Demonstration Schritt für Schritt vorgehen. Interessant ist hier die Nutzung des sogenannten Spread-Operators (drei Punkte):

```

const newFlight = {
  ...oldFlight, date: newFlightDate.toISOString()
};

```

Mit diesem Spread-Operator kopiert das Beispiel das Objekt `oldFlight`. Im Zuge dessen setzt es allerdings den Wert von `date` neu. Diese Kombination aus Klonen und Austauschen ist üblich beim Einsatz von `Immutables`.

Analog dazu geht das Beispiel zum Klonen des Arrays vor:

```

const newFlights = [
  newFlight,
  ...oldFlights.slice(1, this.flights.length - 1)
];

```

Hier tauscht das Beispiel den ersten Eintrag gegen `newFlight`, alle anderen Array-Einträge kopiert es mit dem Spread-Operator.

Beachten Sie, dass der Spread-Operator nur eine flache Kopie erzeugt: Child-Objekte werden also nicht geklont. Das ist auch gut so, denn bei einer tiefen Kopie, bei der auch sämtliche Kinder kopiert werden, ändern sich alle Objektreferenzen. Deswegen würde Angular selbst im OnPush-Modus alle Objekte als geändert ansehen und somit sämtliche Komponenten auf Änderungen prüfen.



Mittlerweile existieren einige Bibliotheken, die die Arbeit mit `Immutables` vereinfachen. Ein sehr populärer Vertreter ist `immer` (<https://www.npmjs.com/package/immer>).

## Immutables und Datenbindung

Wie wir in Kapitel 4 beschrieben haben, durchläuft Angular standardmäßig nach dem Ausführen von Events den gesamten Komponentenbaum. Dabei aktualisiert das Framework alle geänderten Property-Bindings.

Im OnPush-Modus kann Angular jedoch beim Einsatz von Immutables zielgerichtet herausfinden, welche Komponenten von einer Änderung betroffen sind. Hierzu vergleicht es lediglich die Objektreferenzen der an Child-Komponenten per Property-Bindings weitergereichten Werte. Hat sich mindestens eine Objektreferenz geändert, betrachtet es die Child-Komponente näher. Ansonsten wird die Child-Komponente einschließlich ihrer Kinder ignoriert.

Hierzu ist lediglich die Eigenschaft `changeDetection` im Component-Dekorator auf `OnPush` zu setzen (siehe Beispiel 13-4).

*Beispiel 13-4: OnPush für FlightCardComponent aktivieren*

```
// src/app/flight-card/flight-card.component.ts

// Import hinzufügen:
import { ChangeDetectionStrategy } from '@angular/core';
[...]

@Component({
  selector: 'flight-card',
  templateUrl: './flight-card.component.html',
  styleUrls: ['./flight-card.component.scss'],

  // OnPush festlegen:
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class FlightCardComponent implements OnInit, OnChanges {
  [...]
}
```

Wenn Sie nun die Anwendung starten, erneut nach Flügen suchen und danach den ersten Flug verzögern, sollten Sie auch nur den betroffenen Flug aufblinken sehen. Das zeigt, dass Angular jetzt zielgerichtet den geänderten Flug erkannt und alle nicht betroffenen Komponenten ignoriert hat.

Bitte beachten Sie aber auch, dass im OnPush-Modus direkte Änderungen an Eigenschaften nicht erkannt werden. Sie müssen also tatsächlich Objektreferenzen austauschen. Der nächste Abschnitt zeigt eine weitere Möglichkeit, wie Angular im OnPush-Modus Änderungen erkennen kann.

## Observables und Datenbindung

Neben Immutables kann Angular im OnPush-Modus auch Observables nutzen, um sich über Änderungen informieren zu lassen. Hierzu ist das Observable ledig-

lich mit der `async`-Pipe im Template zu binden. Dieses Vorgehen möchten wir hier anhand einer modifizierten Variante des `FlightService` demonstrieren.

Zunächst erhält die abstrakte Basisklasse `FlightService` ein Observable `flights$` und eine `load`-Methode. Außerdem verlagern wir bei dieser Gelegenheit auch gleich mal die `delay`-Methode in den `FlightService` (siehe Beispiel 13-5).

*Beispiel 13-5: FlightService mit Observable*

```
// src/app/flight.service.ts

import { Injectable } from '@angular/core';

// Hinzufügen:
import { Observable } from 'rxjs';
import { DefaultFlightService } from './default-flight.service';
import { Flight } from './flight';

@Injectable({
  providedIn: 'root',
  // Diese Umleitung hinzufügen:
  useClass: DefaultFlightService
})
export abstract class FlightService {

  // Hinzufügen:
  abstract readonly flights$: Observable<Flight[]>;
  abstract load(from: string, to: string): void;
  abstract find(from: string, to: string): Observable<Flight[]>

}
```

Die neue Methode `load` soll mit der bereits existierenden Methode `find` Flüge laden und diese über das Observable `flights$` anbieten.

Da der `DummyFlightService` den `FlightService` implementiert, müssen wir ihm die beiden neuen Member spendieren (siehe Beispiel 13-6).

*Beispiel 13-6: DummyFlightService mit Observable*

```
// src/app/dummy-flight.service.ts

import { Injectable } from '@angular/core';
import { Observable, of } from 'rxjs';
import { Flight } from './flight';
import { FlightService } from './flight.service';

@Injectable()
export class DummyFlightService implements FlightService {

  // Hinzufügen:
  readonly flights$: Observable<Flight[]> = of([]);

  constructor() { }

  // Hinzufügen:
```

```

load(from: string, to: string): void {
    // Kein Inhalt
}

delay(): void {
    // kein Inhalt
}

find(from: string, to: string): Observable<Flight[]> {
    return of([
        { id: 1, from: 'Frankfurt', to: 'Flagranti', date: '2022-01-02T19:00+01:00' },
        { id: 2, from: 'Frankfurt', to: 'Kognito', date: '2022-01-02T19:30+01:00' },
        { id: 3, from: 'Frankfurt', to: 'Mallorca', date: '2022-01-02T20:00+01:00' }
    ]);
}
}

```

Der Einfachheit halber bleiben die Methoden `load` und `delay` beim `DummyFlightService` leer. Beim `DefaultFlightService` werden wir sie jedoch mit Leben füllen. Dieser erhält zunächst das Observable `flights$` sowie ein `BehaviorSubject`, das als Quelle für `flights$` dient (siehe Beispiel 13-7).

*Beispiel 13-7: Änderungen über ein Subject bekannt geben*

```

// src/app/default-flight.service.ts

// Hinzufügen:
import { BehaviorSubject, Observable } from 'rxjs';

[...]

@Injectable()
export class DefaultFlightService implements FlightService {

    // Subject + Observable einfügen:
    private flightSubject = new BehaviorSubject<Flight[]>([]);
    readonly flights$: Observable<Flight[]> = this.flightSubject.asObservable();

    [...]

    // Methoden load und delay in den nächsten Listings
}

```

Die Methode `load` delegiert an `find`, um Flüge zu laden. Diese platziert sie im `BehaviorSubject` (siehe Beispiel 13-8).

*Beispiel 13-8: Die Methode load im DefaultFlightService*

```

// src/app/default-flight.service.ts
[...]

load(from: string, to: string): void {
    this.find(from, to).subscribe(
        flights => {
            this.flightSubject.next(flights);
        }
    );
}

```

```

    },
    error => {
      console.error('error', error);
    }
  );
}

```

Die Methode `delay` entspricht der weiter oben diskutierten Implementierung. Allerdings greift sie nun auch auf die Flüge im `BehaviorSubject` zu (siehe Beispiel 13-9).

*Beispiel 13-9: Die Methode `delay` im `DefaultFlightService`*

```
// src/app/default-flight.service.ts
[...]

delay(): void {
  const oldFlights = this.flightSubject.getValue();
  const oldFlight = oldFlights[0];
  const oldFlightDate = new Date(oldFlight.date);

  const newFlightDate = new Date(oldFlightDate.getTime() + 1000 * 60 * 15);

  const newFlight = {
    ...oldFlight, date: newFlightDate.toISOString()
  };

  const newFlights = [
    newFlight, ...oldFlights.slice(1)
  ];

  this.flightSubject.next(newFlights);
}
```

Nun können wir die eingeführten Ergänzungen in der `FlightSearchComponent` nutzen. Da wir jetzt mit Observables arbeiten, können wir sie auch auf `OnPush` setzen (siehe Beispiel 13-10).

*Beispiel 13-10: `FlightSearchComponent` mit Observable für Flüge*

```
// src/app/flight-search/flight-search.component.ts
[...]

@Component({
  selector: 'app-flight-search',
  templateUrl: './flight-search.component.html',
  styleUrls: ['./flight-search.component.scss'],
  // Hinzufügen:
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class FlightSearchComponent implements OnInit {

  from = 'Hamburg';
  to = 'Graz';
}
```

```

// Entfernen:
// flights: Array<Flight> = [];

// Hinzufügen:
flights$ = this.flightService.flights$;

[...]

// FlightService importieren (falls nicht ohnehin schon der Fall)
constructor(private flightService: FlightService) {
}

ngOnInit(): void {
}

search(): void {

    // Entfernen:
    // this.flightService.find(this.from, this.to).subscribe({
    // [...]
    // });

    // Hinzufügen:
    this.flightService.load(this.from, this.to);

}

select(f: Flight): void {
    this.selectedFlight = f;
}

delay(): void {
    // An Service delegieren:
    this.flightService.delay();
}

}

```

Als Nächstes ist das Template auf flights\$ umzustellen. Beim Iterieren der Flüge nutzen wir die `async`-Pipe:

```

<!-- src/app/flight-search/flight-search.component.html -->
[...]

<div class="row">
    <div *ngFor="let f of flights$ | async"
        class="col-xs-12 col-sm-6 col-md-4 col-lg-4 col-xl-3">
        <flight-card [item]="f" [(selected)]="basket[f.id]">
        </flight-card>
    </div>
</div>

[...]

```

Da wir für das Anzeigen der Schaltflächen das Flug-Array mehrfach benötigen, packen wir es mit einer Kombination aus `ngIf`, `async` und `as` aus dem Observable aus:

```

<!-- src/app/flight-search/flight-search.component.html -->
[...]

<div class="form-group" *ngIf="flights$ | async as flights">
  <button class="btn btn-default" (click)="search()" [disabled]="!f?.valid">
    Search
  </button>

  <button *ngIf="flights.length > 0" class="btn btn-default" (click)="delay()">
    Delay 1st Flight
  </button>
</div>

[...]

```

Wenn Sie nun die Anwendung erneut starten, sollten Sie sehen, dass alles wie früher funktioniert, obwohl nun auch die `FlightSearchComponent` `OnPush` verwendet.

## Immutables und/oder Observables

In den letzten Abschnitten haben wir die `FlightSearchComponent` und die `FlightCardComponent` auf `OnPush` umgestellt. In einem Fall haben wir Immutables verwendet, um Angular im `OnPush`-Modus über Änderungen zu informieren, im anderen Fall haben wir das Gleiche mit Observables gemacht.

Somit könnte man den Eindruck gewinnen, dass man nur eines dieser beiden Konstrukte benötigt. Tatsächlich unterstützen sich Immutables und Observables, so dass sie in der Regel gemeinsam zum Einsatz kommen. Ein Observable informiert Angular über eine Änderung in einer Komponente. Das ist jedoch erst die halbe Miete, denn Angular muss auch wissen, welche Child-Komponenten näher zu betrachten sind.

Ohne `OnPush` in den Child-Komponenten müsste Angular jede einzelne Child-Komponente auf Änderungen prüfen. Haben wir hingegen `OnPush` aktiviert, findet Angular dank Immutables heraus, welche Child-Komponenten es näher zu betrachten gilt.

## Manuelle Änderungsverfolgung

Der Vollständigkeit halber möchten wir hier noch erwähnen, dass Sie Angular auch anweisen können, eine Komponente und deren Child-Komponenten auf Änderungen zu prüfen. Dazu können Sie per Dependency Injection in den einzelnen Komponenten die `ChangeDetectorRef` aus dem Namensraum `@angular/core` injizieren:

```
constructor(private cd: ChangeDetectorRef) { }
```

Danach können Sie bei Bedarf mit der Methode `markForCheck` die Ausführung der Änderungsverfolgung anfordern:

```
this.cd.markForCheck();
```

Diese manuelle Änderungsverfolgung sollte jedoch nur als letzter Ausweg zum Einsatz kommen. In den meisten Fällen kommt man auch mit Immutables und Observables sehr gut über die Runden.

## Lazy Loading von Routen

Standardmäßig werden beim Aufruf einer Angular-Anwendung sämtliche Module geladen. Gerade bei großen Anwendungen führt das zu einer schlechten Startperformance. Der Router bietet hierfür Abhilfe, indem er die Möglichkeit bietet, einzelne Module erst bei Bedarf zu laden.

### Routen für das Lazy Loading einrichten

Um das Lazy Loading zu nutzen, kommen Routen mit der Eigenschaft `loadChildren` zum Einsatz (siehe Beispiel 13-11).

*Beispiel 13-11: Lazy Route für FlightBookingModule*

```
// src/app/app.routes.ts

[...]

export const APP_ROUTES: Routes = [
  [...]
  {
    path: 'flight-booking',
    loadChildren: () => import('./flight-booking/flight-booking.module')
      .then(m => m.FlightBookingModule)
  },
  [...]
  // Wichtig: Diese Route MUSS die LETZTE sein:
  {
    path: '**',
    component: NotFoundComponent
  }
];
```

Die Eigenschaft `loadChildren` verweist auf einen Lambda-Ausdruck, der das gewünschte Angular-Modul bei Bedarf lädt und mit einem Promise retourniert. In der Regel kommt hierfür ein dynamischer Import zum Einsatz. Dieser lädt ein TypeScript-Modul. Das gezeigte `then` bildet dieses TypeScript-Modul auf ein davon exportiertes Angular-Modul ab.

Nun stellt sich die Frage, welche Route Angular aktiviert, wenn der Pfad `/flight-booking` zum Einsatz kommt. Die Antwort darauf liefert die Routenkonfiguration des `FlightBookingModule`. Die dort definierten Pfade sind nun nämlich an `/flight-booking` anzuhängen.

Stellen wir uns vor, wir änderten die Routenkonfiguration des `FlightBookingModule` wie in Beispiel 13-12 gezeigt ab.

*Beispiel 13-12: Routenkonfiguration des lazy FlightBookingModule*

```
// src/app/flight-booking/flight-booking.routes.ts  
[...]  
  
export const FLIGHT_BOOKING_ROUTES: Routes = [  
  {  
    // Alt:  
    // path: 'flight-booking',  
    // Neu:  
    path: '',  
    component: FlightBookingComponent,  
    children: [  
      {  
        // Standardroute  
        path: '',  
        redirectTo: 'flight-search',  
        pathMatch: 'full'  
      },  
      {  
        path: 'flight-search',  
        component: FlightSearchComponent  
      },  
      {  
        path: 'passenger-search',  
        component: PassengerSearchComponent  
      },  
      {  
        path: 'flight-edit/:id',  
        component: FlightEditComponent  
      }  
    ]  
  },  
];
```

Nun würde ein Aufruf von `/flight-booking` aufgrund der festgelegten Standardroute zu einer Umleitung auf `/flight-booking/flight-search` führen. Dieser Pfad würde wiederum zur `FlightSearchComponent` führen. Analog dazu lässt sich mit `/flight-booking/passenger-search` die `PassengerSearchComponent` aktivieren und mit `/flight-booking/flight-edit/myId` die `FlightEditComponent`.

Damit Lazy Loading funktioniert, müssen Sie darauf achten, dass kein anderer Programmteil das `FlightBookingModule` direkt verwendet. Würde beispielsweise das  `AppModule` direkt darauf verweisen, würde die Angular CLI daraus schließen, dass die Anwendung das `FlightBookingModule` von Anfang an benötigt, und es deswegen ins Hauptbundle aufnehmen. Damit lazy Module jedoch bei Bedarf geladen werden können, müssen sie in eigenen Bundles platziert werden.

Deswegen haben wir auch den Import des `FlightBookingModule` in Beispiel 13-13 auskommentiert.

*Beispiel 13-13: Lazy Module dürfen nicht importiert werden!*

```
// src/app/app.module.ts  
[...]
```

```

@NgModule({
  imports: [
    RouterModule.forRoot(APP_ROUTES),
    HttpClientModule,
    BrowserModule,
    // Entfernen - würde Lazy Loading verhindern!!
    // FlightBookingModule
  ],
  ...
})
export class AppModule { }

```

## Lazy Loading im Browser nachvollziehen

Ob das konfigurierte Lazy Loading funktioniert, lässt sich anhand der Konsolenausgabe von `ng serve` erkennen. In diesem Fall sollten Sie dort ein eigenes Bundle für das lazy Modul entdecken.

Außerdem können Sie zur Laufzeit im Browser prüfen, ob dieses Bundle tatsächlich erst bei Bedarf geladen wird. Hierzu nutzen Sie das Registerblatt *Network* in den Dev-Tools von Chrome (siehe Abbildung 13-1). Es zeigt, wann Chrome welche Dateien lädt. Funktioniert das Lazy Loading, lädt Angular das abgespaltene Bundle erst, wenn es benötigt wird.

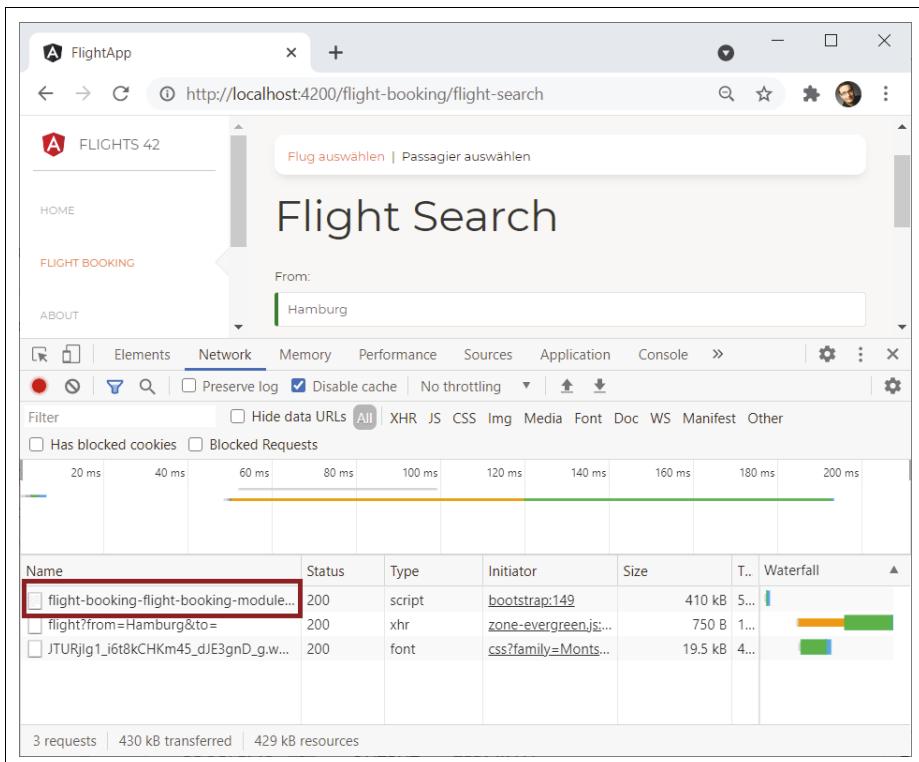


Abbildung 13-1: Lazy Loading in den Dev-Tools nachvollziehen

## Lazy Loading und Tree-Shakable Provider

Bis jetzt haben wir Tree-Shakable Provider lediglich für den Root-Scope eingerichtet:

```
// src/app/flight.service.ts
[...]

@Injectable({
  providedIn: 'root',
  useClass: DefaultFlightService
})
export abstract class FlightService {
  [...]
}
```

Da der Root-Scope für die gesamte Anwendung gilt, lädt Angular solche Services beim Programmstart. Benötigt jedoch nur ein einziges lazy Modul einen bestimmten Service, bietet es sich an, diesen erst mit dem Modul mitzuladen. Hierzu können Sie das Modul der Wahl mit `providedIn` referenzieren:

*Beispiel 13-14: Zyklus beim Einsatz von `providedIn`*

```
// src/app/flight.service.ts
[...]

@Injectable({
  // Lazy Modul referenzieren.
  // Vorsicht - dieses Vorgehensweise führt zu einem Zyklus!!
  providedIn: FlightBookingModule,
  useClass: DefaultFlightService
})
export abstract class FlightService {
  [...]
}
```

Leider führt diese Vorgehensweise zu zyklischen Abhängigkeiten zwischen den einzelnen Programmdateien. Das verhindert, dass die Anwendung ordnungsgemäß funktioniert. Abbildung 13-2 veranschaulicht diesen Zyklus.

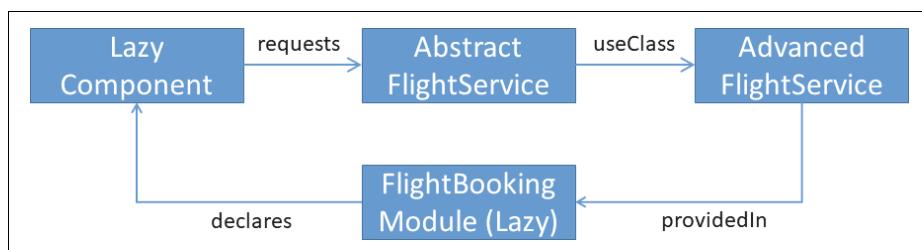


Abbildung 13-2: Problem mit Tree-Shakable Provider und Lazy Loading

Um aus dem Zyklus auszubrechen, benötigen wir ein weiteres Modul. Wir sprechen hierbei auch von einem Api-Modul, das im Ordner des lazy Moduls abgelegt wird. Lassen wir nun sowohl die Services als auch das lazy Modul darauf verweisen, wird der Zyklus unterbrochen (siehe Abbildung 13-3).

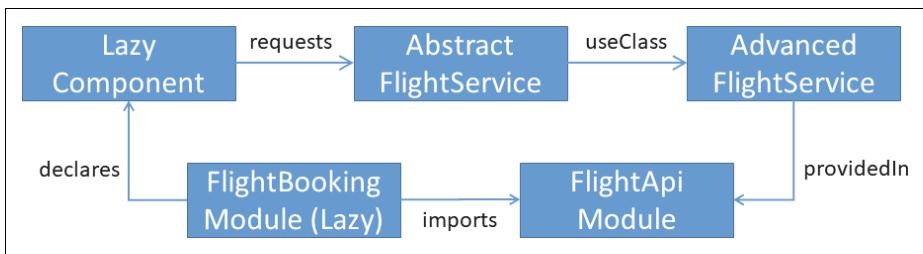


Abbildung 13-3: Zyklische Abhängigkeiten mit ApiModule vermeiden

Das Api-Modul ist lediglich ein Dummy ohne Einträge. Wichtig ist jedoch, dass es in einer eigenen Datei platziert wird (siehe Beispiel 13-15).

*Beispiel 13-15: ApiModule für das lazy FlightBookingModule*

```
// src/app/flight-booking/flight-booking-api.module.ts

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

@NgModule({
  imports: [],
  exports: [],
  declarations: [],
  providers: [],
})
export class FlightBookingApiModule { }
```

Das FlightBookingModule muss nun das FlightBookingApiModule importieren (siehe Beispiel 13-16).

*Beispiel 13-16: Api-Module referenzieren*

```
// src/app/flight-booking/flight-booking.module.ts

[...]
// Hinzufügen:
import { FlightBookingApiModule } from './flight-booking-api.module';

@NgModule({
  imports: [
    // Hinzufügen:
    FlightBookingApiModule,
    [...]
  ],
  ...
})
export class FlightBookingModule { }
```

Die Services, die erst mit dem lazy Modul zu laden sind, können Sie nun für das Api-Modul registrieren (siehe Beispiel 13-17).

*Beispiel 13-17: Der Lazy Service verweist auf das Api-Modul.*

```
// src/app/flight.service.ts  
[...]  
  
import { FlightBookingApiModule } from './flight-booking-api.module';  
  
@Injectable({  
  providedIn: 'root',  
  // Auf das Api-Modul des lazy Moduls verweisen:  
  useClass: FlightBookingApiModule  
})  
export abstract class FlightService {  
  [...]  
}
```

Nun sollte der Service erst mit dem lazy Modul geladen werden.

## Lazy Loading, klassische Provider und Shared Modules

Dieser Abschnitt beschreibt eine gefährliche Konstellation, die sich bei der Kombination aus Lazy Loading und klassischen Providern in geteilten Modulen ergibt. Diese Konstellation kommt beim Einsatz von Tree-Shakable Providern erst gar nicht zustande. Aus diesem Grund empfehlen wir generell deren Nutzung.

Um diese Konstellation zu veranschaulichen, führen wir in diesem Abschnitt einen AuthService ein, der den Namen des aktuellen Benutzers verwaltet (siehe Abbildung 13-4).

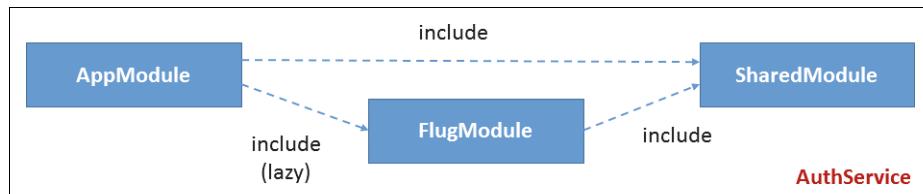


Abbildung 13-4: AuthService in SharedModule

Sowohl das AppModule als auch das lazy FlightBookingModule referenzieren das Shared Module, das den AuthService bereitstellt.

Genau darin liegt das Problem, denn Angular richtet lazy Modulen einen eigenen Dependency-Injection-Scope ein. Deswegen entstehen in der gezeigten Konstellation auch zwei Instanzen des AuthService: eine im Scope von FlightBookingModule und eine im globalen Root-Scope (siehe Abbildung 13-5).

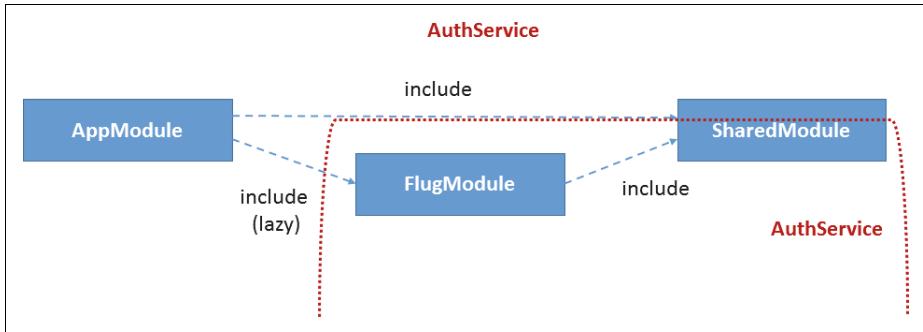


Abbildung 13-5: Mehrere Namensräume bei Lazy Loading und Shared Modules

Bevor wir Ihnen eine Lösung für dieses Dilemma zeigen, beschreiben wir noch im nächsten Abschnitt die Implementierung und Nutzung des AuthService.

### Den AuthService implementieren und nutzen

Zunächst wird der AuthService mit der Angular CLI im Ordner *shared* generiert:

```
ng g s shared/auth/auth
```

Die Implementierung des AuthService gestaltet sich wie in Beispiel 13-18.

#### Beispiel 13-18: Implementierung des AuthService

```
// src/app/shared/auth/auth.service.ts

import { Injectable } from '@angular/core';

@Injectable()
export class AuthService {

  userName: string | null = null;

  constructor() { }

  login(userName: string): void {
    // Authentifizierung für ehrliche Benutzer
    this.userName = userName;
  }

  logout(): void {
    this.userName = null;
  }
}
```

Wie bereits eingangs angemerkt, bietet das SharedModule den AuthService an. Dazu weist es einen entsprechenden Provider auf (siehe Beispiel 13-19).

*Beispiel 13-19: SharedModule mit Provider für den AuthService*

```
// src/app/shared/shared.module.ts

[...]
// Hinzufügen:
import { AuthService } from './auth/auth.service';

@NgModule({
  ...
  // Hinzufügen:
  providers: [
    AuthService
  ],
  ...
})
export class SharedModule { }
```

Im Weiteren gehen wir davon aus, dass das SharedModule ebenfalls für das AppModule interessant ist. Deswegen listet auch das AppModule das SharedModule in seinem imports-Array:

*Beispiel 13-20: SharedModule importieren*

```
// src/app/app.module.ts

// Hinzufügen:
import { SharedModule } from './shared/shared.module';

@NgModule({
  imports: [
    // Hinzufügen:
    SharedModule,
    [...]
  ],
  ...
})
export class AppModule { }
```

Auch das per Lazy Loading bezogene FlightBookingModule importiert das SharedModule. Falls Sie die Beispiele der vorangegangenen Kapitel umgesetzt haben, sollte das jedoch schon der Fall sein (siehe Beispiel 13-21).

*Beispiel 13-21: Auch das FlightBookingModule referenziert das SharedModule.*

```
// src/app/flight-booking/flight-booking.module.ts

[...]
import { SharedModule } from '../shared/shared.module';
[...]

@NgModule({
  imports: [
    // Sollte schon da sein:
    SharedModule
```

```

  ],
  ...
})
export class FlightBookingModule { }

```

Um den Benutzern das Anmelden zu erlauben, lässt sich die HomeComponent den AuthService injizieren (siehe Beispiel 13-22).

*Beispiel 13-22: Die HomeComponent ermöglicht über den AuthService eine Authentifizierung.*

```
// src/app/home/home.component.ts
```

```

import { Component, OnInit } from '@angular/core';
import { AuthService } from '../shared/auth/auth.service';

@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.scss']
})
export class HomeComponent implements OnInit {

  get userName(): string | null {
    return this.authService.userName;
  }

  constructor(private authService: AuthService) { }

  ngOnInit(): void {
  }

  login(): void {
    this.authService.login('Max');
  }

  logout(): void {
    this.authService.logout();
  }
}

```

Außerdem stellt die HomeComponent Methoden für das An- und Abmelden zur Verfügung. Diese Methoden delegieren an den AuthService ebenso wie der Getter für den Benutzernamen (userName).

Das Template der HomeComponent bindet sich an die neuen Member:

```

<!-- src/app/home/home.component.html -->

<h1 *ngIf="!userName">Welcome!</h1>
<h1 *ngIf="userName">Welcome, {{userName}}!</h1>

<button class="btn btn-default" (click)="login()">Login</button>
<button class="btn btn-default" (click)="logout()">Logout</button>

```

Die PassengerSearchComponent aus dem lazy FlightBookingModule nutzt ebenfalls den AuthService. Sie ruft den aktuellen Benutzernamen ab und zeigt ihn an (siehe Beispiel 13-23).

*Beispiel 13-23: Auch die PassengerSearchComponent nutzt den AuthService.*

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

// Hinzufügen:
import { AuthService } from 'src/app/shared/auth/auth.service';

@Component({
  selector: 'app-passenger-search',
  templateUrl: './passenger-search.component.html',
  styleUrls: ['./passenger-search.component.scss']
})
export class PassengerSearchComponent implements OnInit {

  // Hinzufügen:
  userName = this.authService.userName;

  // Hinzufügen:
  constructor(private authService: AuthService) { }

  ngOnInit(): void {
  }
}
```

Das Template der PassengerSearchComponent zeigt den Benutzernamen an:

```
<!-- src/app/flight-booking/passenger-search/passenger-search.component.html -->

<h1>Passenger</h1>
<p>Name: {{userName}}</p>
```

Beim Testen der gezeigten Implementierung fällt auf, dass sich der Benutzer zwar anmelden kann (siehe Abbildung 13-6), die Passagiersuche seinen Benutzernamen aber nicht kennt (siehe Abbildung 13-7).

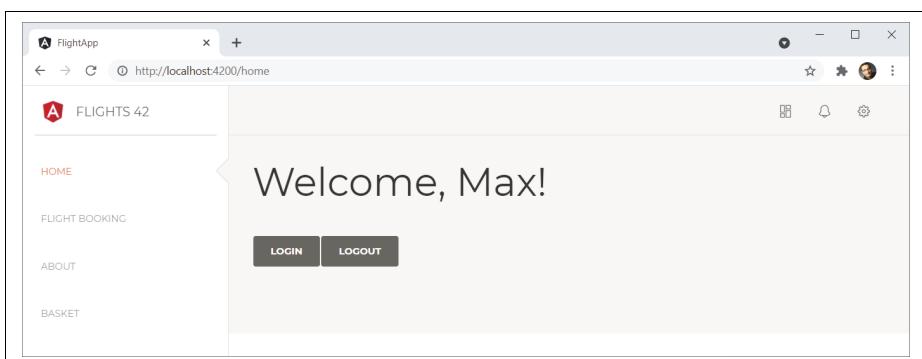


Abbildung 13-6: Demoanwendung nach der Anmeldung

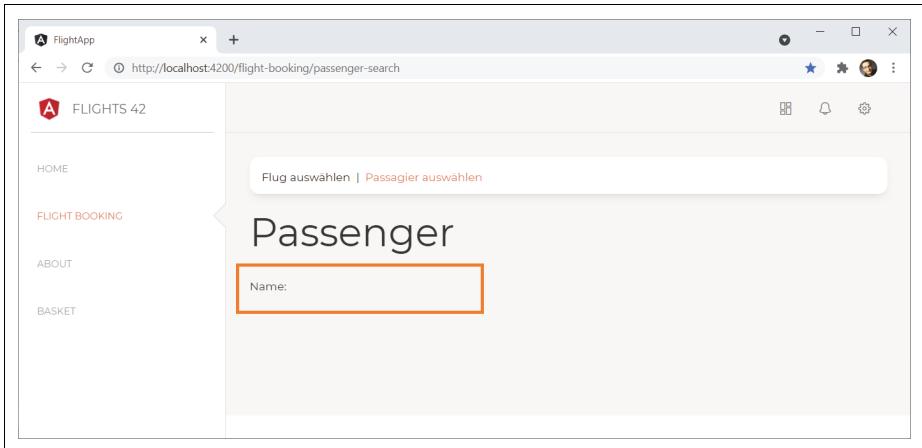


Abbildung 13-7: Passagiersuche kennt trotz Anmeldung den Benutzernamen nicht.

Wie wir bereits eingangs erläutert haben, liegt das daran, dass Angular für lazy Module einen eigenen Scope einrichtet. Somit entstehen im Laufe der Zeit zwei Instanzen von AuthService. Während im betrachteten Fall die Authentifizierung mit dem einen AuthService erfolgt, nutzt die PassengerSearchComponent im nachgeladenen FlightBookingModule die andere Instanz. Bei dieser ist der Benutzer nicht als angemeldet vermerkt.

## Korrekte Nutzung von SharedModules beim Einsatz von Lazy Loading

Zum Umgehen des hier beschriebenen Dilemmas empfiehlt das Angular-Team ein Muster. Dieses sieht vor, dass das SharedModule in zwei Ausprägungen anzubieten ist: einmal mit Providern, die global gelten, und einmal ohne (siehe Abbildung 13-8).

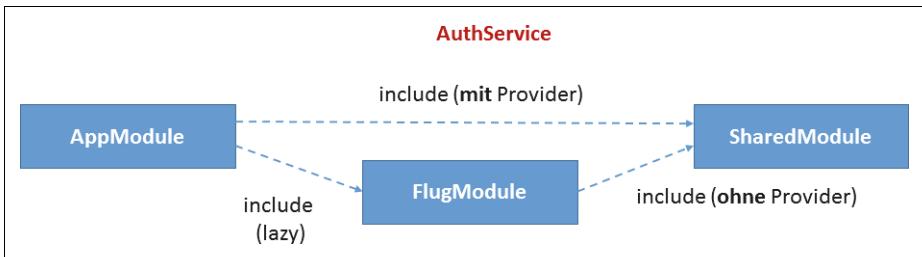


Abbildung 13-8: Korrekte Nutzung eines SharedModule zusammen mit Lazy Loading

Die Implementierung dieses Ansatzes ist einfacher, als es auf den ersten Blick scheint. Zunächst müssen Sie das Modul gänzlich ohne Provider einrichten. Zusätzlich geben Sie ihm eine statische Methode, die per definitionem den Namen

forRoot tragen sollte. Diese Methode liefert das Modul inklusive Provider zurück (siehe Beispiel 13-24):

*Beispiel 13-24: Das überarbeitete SharedModule*

```
// src/app/shared/shared.module.ts

// Hinzufügen:
import { ModuleWithProviders } from '@angular/core';

[...]

@NgModule({
  ...
  providers: [
    // Alle Provider hier entfernen!
  ],
  ...
})
export class SharedModule {

  // Hinzufügen:
  static forRoot(): ModuleWithProviders<SharedModule> {
    return {
      ngModule: SharedModule,
      providers: [
        AuthService
      ]
    };
  }
}
```

Die neue forRoot-Methode liefert ein `ModuleWithProviders`-Objekt. Dieses Objekt ergänzt ein bestehendes Modul um Provider. Das Interessante daran ist, dass bereits der Angular-Compiler diese Methode aufruft und auswertet. Das bedeutet, Sie dürfen in solchen Methoden nur Programmcode platzieren, den der Compiler auswerten kann. Mit einer einzigen `return`-Anweisung sind Sie auf der sicheren Seite. Zusätzlich könnten Sie noch übergebene Parameter nutzen, um bestimmte Provider einzurichten:

```
static forRoot(config: SomeConfig): ModuleWithProviders<MyModule> {
  return {
    ngModule: MyModule,
    providers: [
      {
        provide: SomeConfig, useValue: config
      }
    ]
  };
}
```

Das `AppModule`, auch Root-Module genannt, muss nun das `SharedModule` mittels `forRoot` importieren (siehe Beispiel 13-25).

*Beispiel 13-25: Das AppModule nutzt die Methode forRoot, um das SharedModule zu referenzieren.*

```
// src/app/app.module.ts  
[...]  
  
@NgModule({  
    imports: [  
        // Mit forRoot importieren:  
        SharedModule.forRoot(),  
        [...]  
    ],  
    ...  
})  
export class AppModule { }
```

Alle anderen Module können das SharedModule wie gehabt auf klassische Weise referenzieren. Somit richtet Angular die Provider nur einmal auf globaler Ebene für alle Module ein.

Dieses Muster nutzt auch der Router selbst, indem das RouterModule eine Methode forRoot für das Root-Module anbietet. Daneben bietet es auch eine Methode forChild an, die auf die gleiche Weise nur Provider für ein bestimmtes Feature-Module definiert. Diese Provider richten unter anderem die Routenkonfiguration für das Feature-Module ein.



Eine Alternative zum Einsatz von forRoot/forChild ist die Nutzung eines Core-Module. Dabei handelt es sich um ein Modul, das Provider für alle globalen Services anbietet und das per Konvention nur ins AppModule importiert werden darf. Somit wird auch sichergestellt, dass Angular die globalen Services nur im Root-Scope einrichtet.

Allerdings raten wir vom Einsatz eines Core-Module ab, da es schnell zur »allwissenden Müllhalde« wird. Außerdem bricht es durch das Sammeln sämtlicher globaler Services die Kapselung auf, die sich durch eine gute Modularisierung ergeben würde.

## Preloading

Preloading geht über die Möglichkeiten von Lazy Loading hinaus und erlaubt weitere Performanceoptimierungen: Es nutzt freie Ressourcen nach dem Start der Anwendung, um Module nachzuladen, die später per Lazy Loading angefordert werden könnten. Wenn der Router diese Module später tatsächlich benötigt, stehen sie augenblicklich zur Verfügung.

### Preloading aktivieren

Um das Preloading zu aktivieren, übergeben Sie an RouterModule.forRoot() eine PreloadingStrategy (siehe Beispiel 13-26).

### Beispiel 13-26: PreloadingStrategy registrieren

```
// src/app/app.module.ts

[...]

// Hinzufügen:
import { PreloadAllModules } from '@angular/router';

@NgModule({
  imports: [
    [...]
    // preloadingStrategy registrieren:
    RouterModule.forRoot(APP_ROUTES, {preloadingStrategy: PreloadAllModules}),
    [...]
  ],
  ...
})
export class AppModule { }
```

Die hier verwendete Strategie PreloadAllModules ist in Angular inkludiert und lädt sämtliche lazy Module bereits nach dem Programmstart. Die Anwendung wird also zuerst ohne lazy Module gestartet und somit rascher präsentiert. Unmittelbar danach beginnt die Strategie jedoch mit dem Nachladen sämtlicher lazy Module, sodass sie sehr wahrscheinlich zur Verfügung stehen, wenn sie benötigt werden. Aktiviert der Benutzer die lazy Route, bevor die Preloading-Strategie sie laden konnte, findet klassisches Lazy Loading statt.

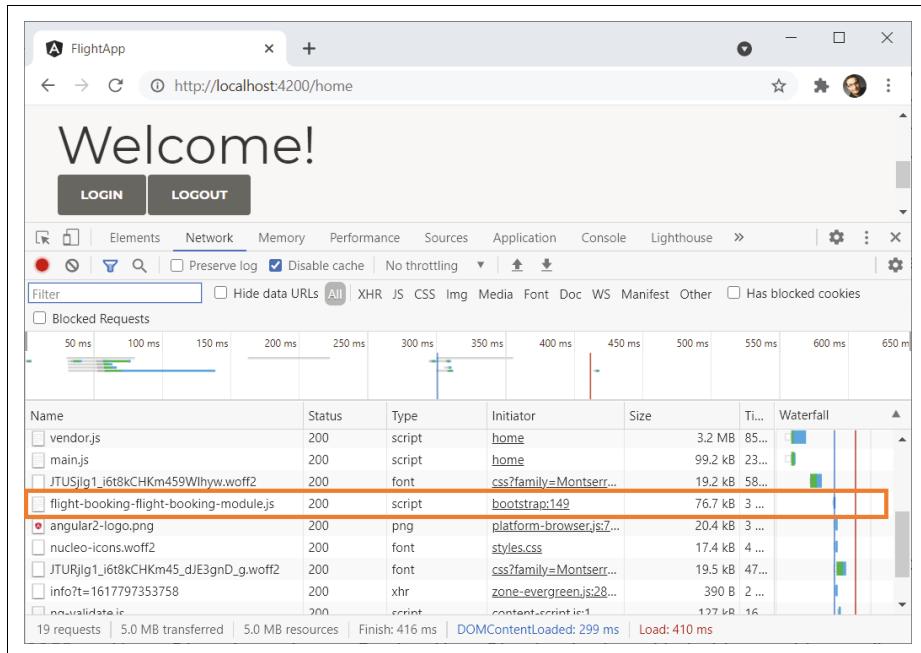


Abbildung 13-9: Angular lädt Module nach dem Start der Anwendung vor.

Das Ergebnis dieses Unterfangens können Sie in Chrome mit dem Registerblatt *Network* in den Dev-Tools beobachten. Hier sollten Sie sehen, dass das lazy Modul vorgeladen wird (siehe Abbildung 13-9).

## Eigene Preloading-Strategien entwickeln

Neben der von Angular angebotenen `PreloadAllModules`-Strategie können Sie auch eigene Strategien implementieren. Auf diese Weise geben Sie vor, wann welche lazy Module vorzuladen sind. Dazu stellen Sie einen Service bereit, der das Interface `PreloadingStrategy` implementiert:

*Beispiel 13-27: Benutzerdefinierte Preloading-Strategie*

```
// src/app/shared/custom-preloading.strategy.ts

import { Injectable } from '@angular/core';
import { PreloadingStrategy, Route } from '@angular/router';
import { Observable, of } from 'rxjs';
import { delay, mergeMap } from 'rxjs/operators';

@Injectable({
  providedIn: 'root'
})
export class CustomPreloadingStrategy implements PreloadingStrategy {

  constructor() { }

  preload(route: Route, fn: () => Observable<any>): Observable<any> {
    return of(true).pipe(delay(7000), mergeMap(_ => fn()));
  }
}
```



In den meisten Fällen lohnt es sich, mit der `PreloadAllModules`-Strategie zu starten. Eigene Strategien sind eher ein Maßnahme für das Finetuning, die Sie später bei Bedarf nutzen können.

Das Interface `PreloadingStrategy` gibt die Methode `preload` vor. Sie nimmt eine Beschreibung einer Route, die Angular vorladen könnte, entgegen. Der zweite Parameter ist eine Funktion, die das Preloading anstößt. Sie können nun innerhalb dieser Methode entscheiden, ob bzw. wann Sie diese Methode ausführen. Beispielsweise könnten Sie abwarten, bis sich der Benutzer angemeldet hat und dann anhand der zugewiesenen Rollen und Rechte entscheiden, was Sie vorladen wollen.

Das hier gezeigte Experiment verzögert das Preloading um sieben Sekunden. Um die gezeigte `PreloadingStrategy` auszuprobieren, müssen Sie sie im `AppModule` an `RouterModule.forRoot` übergeben (siehe Beispiel 13-28).

*Beispiel 13-28: Eigene Preloading-Strategie registrieren*

```
// src/app/app.module.ts

[...]
import { CustomPreloadingStrategy } from './shared/custom-preloading.strategy';

@NgModule({
  imports: [
    [...]
    // CustomPreloadingStrategy registrieren:
    RouterModule.forRoot(APP_ROUTES, {preloadingStrategy: CustomPreloadingStrategy}),
    [...]
  ],
  ...
})
export class AppModule { }
```

## Selektives Preloading mit eigener Preloading-Strategie

Als Beispiel für eine weitere Preloading-Strategie zeigt dieser Abschnitt eine Implementierung, die das Preloading auf bestimmte Module beschränkt. Dazu erhalten die gewünschten Routen eine benutzerdefinierte Eigenschaft `preload` (siehe Beispiel 13-29).

*Beispiel 13-29: Routenkonfiguration mit eigenen Eigenschaften*

```
// src/app/app.routes.ts
[...]

export const APP_ROUTES: Routes = [
  [...]
  {
    path: 'flight-booking',
    loadChildren: () => import('./flight-booking/flight-booking.module')
      .then(m => m.FlightBookingModule),
    // Hinzufügen:
    data: {
      preload: true
    }
  },
  [...]
  // Diese Route muss die letzte sein!
  {
    path: '**',
    component: NotFoundComponent
  }
];
```

Die Eigenschaft `data` ist für solche benutzerdefinierten Erweiterungen vorgesehen. Die Preloading-Strategie kann nun prüfen, ob die übergebene Route diese Eigenschaft aufweist (siehe Beispiel 13-30).

*Beispiel 13-30: Eigene PreloadingStrategy für selektives Preloading*

```
// src/app/shared/custom-preloading.strategy.ts

import { Injectable } from '@angular/core';
import { PreloadingStrategy, Route } from '@angular/router';
import { Observable, of } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class CustomPreloadingStrategy implements PreloadingStrategy {

  constructor() { }

  preload(route: Route, fn: () => Observable<any>): Observable<any> {
    if (route.data?.preload) {
      // Preloading anstoßen
      return fn();
    }
    // Kein Preloading anstoßen,
    // Dummy-Observable zurückliefern.
    return of([]);
  }
}
```

In diesem Fall lädt die Preloading-Strategie die Route mit der entgegengenommenen Funktion und liefert das Observable zurück, das sie von der Funktion erhalten hat. Ansonsten gibt sie ein (Dummy-)Observable zurück, das den Wert null transportiert. Das Registrieren der `CustomPreloadingStrategy` erfolgt daraufhin, wie bereits weiter oben beschrieben.

## Zusammenfassung

Auch wenn die Datenbindung bei Angular bereits in der Standardeinstellung sehr effizient ist, lässt sie sich mit der Strategie `OnPush` weiter beschleunigen. In diesem Fall findet Angular zielgerichtet heraus, welche Komponenten zu aktualisieren sind. Damit das möglich ist, müssen Sie die zu bindenden Daten als Immutables und via Observables anbieten.

Zur Beschleunigung der Startgeschwindigkeit bietet der Angular-Router Lazy Loading. Das ermöglicht der Anwendung, einzelne Teile erst bei Bedarf zu laden. Damit diese Teile vorhanden sind, wenn die Anwendung sie benötigt, lässt sich Lazy Loading mit Preloading kombinieren. Über Preloading-Strategien kann angegeben werden, ob bzw. wann welcher Anwendungsteil vorgeladen werden soll.



# Querschnittsfunktionen

Querschnittsfunktionen sind meist technische Anforderungen, die es immer und immer wieder zu berücksichtigen gilt. Beispiele dafür sind unter anderem Authentifizierung, Protokollierung oder die Behandlung von Fehlern. Natürlich möchte man die dafür nötigen Methodenaufrufe nicht ständig wiederholen müssen. Idealerweise werden sie automatisch aktiv.

Dieses Kapitel geht auf Möglichkeiten ein, die Angular dafür bietet: Mit Guards lassen sich Routenwechsel verhindern, und Resolver verzögern die Aktivierung von Routen, bis alle nötigen Daten vorliegen. Außerdem informieren Routerevents über die Tätigkeiten des Routers. Basierend darauf, kann eine Anwendung einen generischen Loading-Indicator ein- und ausblenden.

Neben dem Router bietet auch der *HttpClient* mit den sogenannten *HttpInterceptoren* die Möglichkeit zur Umsetzung von Querschnittsfunktionen. Damit lassen sich sämtliche ausgehenden Anfragen und ankommenden Antworten modifizieren.

Am Ende geht das Kapitel auf Möglichkeiten zur Authentifizierung des Benutzers in Angular-Anwendungen ein. Die hier vorgestellten Ideen fußen teilweise auf *Guards* und *HttpInterceptoren*.

## Guards

Mit Guards können sich Angular-Anwendungen über Routenwechsel informieren lassen. Bei Guards handelt es sich lediglich um Services mit vorgegebenen Methoden, die der Router zu bestimmten Zeitpunkten aufruft. Diese Methoden können auch ins Routing eingreifen: Ihr zurückgelieferter Wert bestimmt, ob der Router den angeforderten Routenwechsel tatsächlich durchführen darf.

Kann die Methode ihre Entscheidung augenblicklich bekannt geben, liefert sie einen Boolean. Um die Entscheidung hinauszuzögern, liefert sie zunächst lediglich ein `Observable<boolean>` oder einen `Promise<boolean>`. Steht die Entscheidung später fest, kann die Methode über diese Mechanismen den Router benachrichtigen. Dieses Vorgehen ist beispielsweise notwendig, wenn zur Entscheidungsfindung eine Web-API zu konsultieren oder Rücksprache mit dem Benutzer zu halten ist.

Für unterschiedliche Arten von Guards definiert Angular auch unterschiedliche Interfaces, die es zu implementieren gilt (siehe Tabelle 14-1).

Tabelle 14-1: Typen für Guards

Interface	Methode	Beschreibung
CanActivate	canActivate	Legt fest, ob die gewünschte Route aktiviert werden darf.
CanActivateChild	canActivateChild	Legt fest, ob bzw. welche Child-Routes einer Route aktiviert werden dürfen.
CanLoad	canLoad	Legt fest, ob ein Modul per Lazy Loading geladen werden darf.
CanDeactivate	canDeactivate	Legt fest, ob eine Route deaktiviert werden darf.

## Das Aktivieren von Routen verhindern

In diesem Abschnitt demonstrieren wir die Nutzung von Guards anhand einer Implementierung, die unberechtigte Benutzer von Routen fernhält. Dazu ist das Interface `CanActivate` zu implementieren. Dies dient weniger der Sicherheit, zumal Sicherheit bei browserbasierten SPAs immer im Backend zu realisieren ist. Vielmehr dient dies der Benutzerfreundlichkeit, da hierdurch die Anwendung den Benutzer bei Bedarf zur Anmeldung auffordern kann.

Der Guard lässt sich mit der CLI erzeugen:

```
ng g guard shared/auth/Auth --implements CanActivate
```

Anschließend können Sie den Guard mit Leben füllen (siehe Beispiel 14-1).

Beispiel 14-1: Mit Guards Routen vor nicht authentifizierten Benutzern schützen

```
// src/app/shared/auth/auth.guard.ts

import { Injectable } from '@angular/core';
import {
  CanActivate,
  ActivatedRouteSnapshot,
  RouterStateSnapshot,
  Router,
  UrlTree
} from '@angular/router';
import { AuthService } from './auth.service';

@Injectable({
  providedIn: 'root'
})
export class AuthGuard implements CanActivate {

  constructor(private authService: AuthService, private router: Router) {}

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean | UrlTree {
    if (!this.authService.isAuthenticated()) {
      this.router.navigate(['/login']);
      return false;
    }
    return true;
  }
}
```

```

        if (this.authService.userName) {
            return true;
        }
        else {
            return this.router.createUrlTree(['/home', { needLogin: true }]);
        }
    }
}

```

Wie Sie hier sehen, handelt es sich bei Guards um Services, die mindestens eines der oben erwähnten Interfaces implementieren. Neben dem in Kapitel 15 beschriebenen AuthService zum Anmelden von Benutzern lässt sich der Guard auch den Router injizieren. Damit leitet er Benutzer mit fehlenden Berechtigungen auf eine andere Route weiter.

Die von CanActivate vorgegebene gleichnamige Methode erhält vom Router einen ActiveRouteSnapshot, der über die gewünschte Route informiert, sowie einen Router StateSnapshot, der über die aktuelle Route Auskunft gibt.

Die Methode canActivate wendet sich an den AuthService, um herauszufinden, ob der aktuelle Benutzer angemeldet ist. Ist dem so, liefert sie true zurück und erlaubt somit die gewünschte Aktivierung. Ansonsten erzeugt sie mit der Methode createUrlTree des Routers eine Datenstruktur, die auf die Route home verweist.

Diesen sogenannten UrlTree liefert sie zurück. Das veranlasst den Router, auf home weiterzuleiten. Über den angehängten Parameter needLogin erfährt die dahinterstehende HomeComponent den Grund für die Umleitung und kann eine entsprechende Meldung ausgeben.

Liefert canActivate hingegen false, bricht der Router den Routenwechsel kommentarlos ab.

Um den Guard für eine Route zu aktivieren, muss sie mit ihrer Eigenschaft canActivate lediglich darauf verweisen (siehe Beispiel 14-2).

#### *Beispiel 14-2: AuthGuard in Routenkonfiguration hinterlegen*

```

// src/app/flight-booking/flight-booking.routes.ts

[...]

// Hinzufügen:
import { AuthGuard } from './shared/auth/auth.guard';

export const FLIGHT_BOOKING_ROUTES: Routes = [
{
    path: '',
    component: FlightBookingComponent,
    children: [
        [...]
        {
            path: 'passenger-search',
            component: PassengerSearchComponent,

```

```

        // Hinzufügen:
        canActivate: [AuthGuard]
    },
    [...]
],
},
];

```

Damit eine Route mehrere solcher Guards nutzen kann, handelt es sich bei canActivate um ein Array. Nur dann, wenn sämtliche Guards eines solchen Arrays true liefern, führt der Router den Routenwechsel durch.

Wenn Sie nun die Anwendung starten, sollten Sie lediglich nach einer Anmeldung die PassengerSearchComponent aktivieren können.

## Das Deaktivieren einer Komponente verhindern

Um die Nutzung von CanDeactivate zu demonstrieren, setzen wir hier einen Guard ein, der den Benutzer fragt, ob er oder sie die Route tatsächlich verlassen möchte. Mit solch einem Verhalten kann eine Anwendung beispielsweise verhindern, dass geänderte, jedoch nicht gespeicherte Daten verloren gehen.

Auch dieser Guard lässt sich mit der CLI generieren:

```
ng g guard shared/exit --implements CanDeactivate
```

Der Guard implementiert das Interface CanDeactivate, und ein Typparameter verweist auf die zu schützende Komponente (siehe Beispiel 14-3).

*Beispiel 14-3: Guard zum Verhindern der Deaktivierung einer Route*

```
// src/app/shared/exit/exit.guard.ts

import { Injectable } from '@angular/core';
import { CanDeactivate, ActivatedRouteSnapshot, RouterStateSnapshot }
  from '@angular/router';
import { Observable } from 'rxjs';

export interface CanExit {
  canExit(): Observable<boolean>;
}

@Injectable({
  providedIn: 'root'
})
export class ExitGuard implements CanDeactivate<CanExit> {
  canDeactivate(
    component: CanExit,
    currentRoute: ActivatedRouteSnapshot,
    currentState: RouterStateSnapshot): Observable<boolean> {

    return component.canExit();
  }
}
```

Damit der Guard nicht nur für eine einzige Komponente tätig werden kann, kommt das Interface `CanExit` zum Einsatz. Dieses Interface muss die zu schützende Komponente implementieren.

Die Methode `canDeactivate` bekommt vom Router als ersten Parameter die Komponente übergeben. Die anderen beiden Parameter informieren analog zum `CanActivate`-Guard über die aktuelle sowie die angeforderte Route. Die Entscheidung darüber, ob die aktuelle Route verlassen werden darf, überlässt `canDeactivate` voll und ganz der Komponente. Dazu delegiert sie an ihre `canExit`-Methode.

Damit wir den Guard mit der `FlightEditComponent` aus Kapitel 8 testen können, muss sie unser Interface `CanExit` implementieren (siehe Beispiel 14-4).

*Beispiel 14-4: Zu schützende FlightEditComponent*

```
// src/app/flight-booking/flight-edit/flight-edit.component.ts  
[...]  
  
// Hinzufügen:  
import { CanExit } from 'src/app/shared/exit/exit.guard';  
import { Subject, Observable } from 'rxjs';  
  
@Component({  
  selector: 'app-flight-edit',  
  templateUrl: './flight-edit.component.html',  
  styleUrls: ['./flight-edit.component.scss']  
})  
export class FlightEditComponent implements OnInit, CanExit {  
  
  [...]  
  
  // Hinzufügen:  
  canExitSubject = new Subject<boolean>();  
  showWarning = false;  
  
  [...]  
  
  canExit(): Observable<boolean> {  
    this.canExitSubject = new Subject<boolean>();  
    this.showWarning = true;  
    return this.canExitSubject;  
  }  
  
  decide(decision: boolean): void {  
    this.showWarning = false;  
    this.canExitSubject.next(decision);  
    this.canExitSubject.complete();  
  }  
  
  [...]  
}
```

Sobald der Guard `canExit` aufruft, setzt die Komponente ihre Eigenschaft `showWarning` auf `true`. Damit blendet sie die Warnmeldung ein. Diese fragt den Benutzer, ob er tatsächlich die Route verlassen möchte. Da `canDeactivate` die Antwort des Benutzers abwarten muss, liefert sie das `canExitSubject` zurück.

Die Methode `decide` nimmt die Benutzerentscheidung entgegen und blendet die Warnmeldung wieder aus, indem sie die Eigenschaft `showWarning` auf `false` zurücksetzt. Danach gibt sie über das `canExitSubject` die Benutzerentscheidung an den Router weiter. Außerdem muss `decide` die `complete`-Methode des Subjects anstoßen, da der Router erst danach aktiv wird.

Das Template der `FlightEditComponent` blendet die Warnmeldung in Abhängigkeit von `showWarning` ein (siehe Beispiel 14-5).

*Beispiel 14-5: Template für eine Warnung*

```
<!-- src/app/flight-booking/flight-edit/flight-edit.component.html -->
[...]

<div *ngIf="showWarning" class="alert alert-warning">
  <div>Daten wurden nicht gespeichert! Trotzdem Maske verlassen?</div>
  <div>
    <a href="javascript:void(0)" (click)="decide(true)" class="btn btn-danger">
      Ja
    </a>
    <a href="javascript:void(0)" (click)="decide(false)" class="btn btn-default">
      Nein
    </a>
  </div>
</div>
```

Die Entscheidung des Benutzers, die die Warnung über einen der beiden Links entgegennimmt, delegiert das Template an die Methode `decide`. Nun ist der Guard noch in die Routenkonfiguration einzutragen (siehe Beispiel 14-6).

*Beispiel 14-6: ExitGuard in der Routenkonfiguration registrieren*

```
// src/app/flight-booking/flight-booking.routes.ts

[...]
// Hinzufügen:
import { ExitGuard } from '../shared/exit/exit.guard';

export const FLIGHT_BOOKING_ROUTES: Routes = [
  {
    path: '',
    component: FlightBookingComponent,
    children: [
      [...],
      {
        path: 'flight-edit/:id',
        component: FlightEditComponent,
```

```

        // Hinzufügen:
        canDeactivate: [ExitGuard]
    }
],
},
];

```

Zum Ausprobieren müssen Sie jetzt in der Anwendung nach Flügen suchen und für einen Flug in die Editiermaske wechseln. Wenn Sie die Maske wieder verlassen wollen, sollte die Warnmeldung erscheinen (siehe Abbildung 14-1).

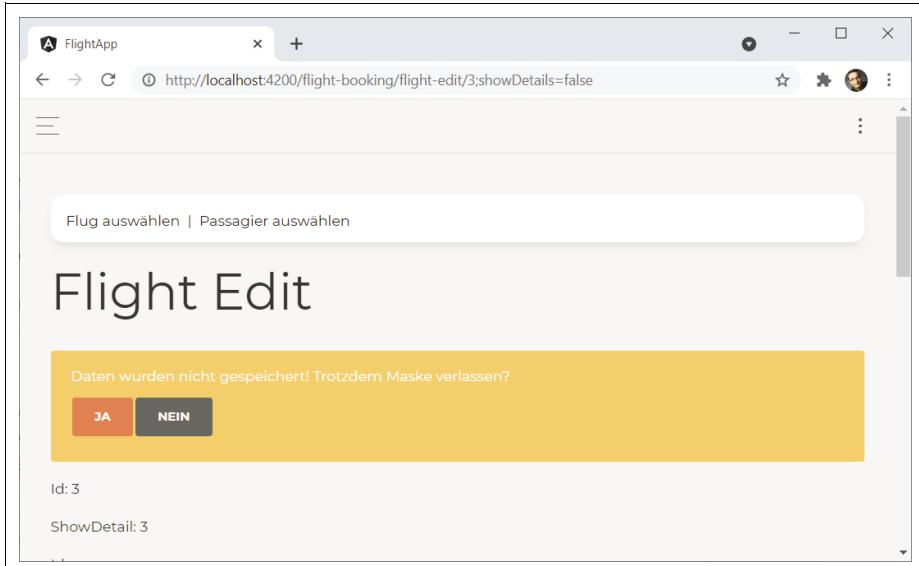


Abbildung 14-1: Der ExitGuard in Aktion

## Events

Damit sich die Anwendung über die Tätigkeiten des Routers auf dem Laufenden halten kann, veröffentlicht der Router Events. Tabelle 14-2 zeigt eine Auswahl an möglichen Events.

Tabelle 14-2: Ausgewählte Routerevents

NavigationStart	Ergebnis
RoutesRecognized	Der Wechsel zu einer neuen Route wurde gestartet.
NavigationEnd	Der Router konnte aus der URL die gewünschte Route ableiten.
NavigationCancel	Der Wechsel zu einer neuen Route wurde abgeschlossen.
NavigationError	Der Wechsel zu einer neuen Route wurde durch einen Guard abgebrochen.

Eine Anwendung kann diese Ereignisse beispielsweise nutzen, um beim Routenwechsel einen Loading-Indicator einzublenden. Die folgenden Ausführungen zeigen, wie sich dies in der AppComponent erledigen lässt.

Diese lässt sich den Router injizieren und informiert sich über das Observable events über auftretende Routerevents (siehe Beispiel 14-7).

*Beispiel 14-7: AppComponent nutzt Routerevents zum Einblenden eines Loading-Indicators.*

```
// src/app/app.component.ts
```

```
import { Component, OnInit } from '@angular/core';
import {
  NavigationCancel,
  NavigationEnd,
  NavigationError,
  NavigationStart,
  Router
} from '@angular/router';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent implements OnInit {
  title = 'Hello World!';
  showWaitInfo = false;

  constructor(private router: Router) { }

  ngOnInit(): void {
    this.router.events.subscribe(event => {
      if (event instanceof NavigationStart) {
        this.showWaitInfo = true;
      }
      else if (event instanceof NavigationEnd
        || event instanceof NavigationCancel
        || event instanceof NavigationError) {
        this.showWaitInfo = false;
      }
    });
  }
}
```

Abhängig vom jeweiligen Event setzt diese Implementierung das Flag `showWaitInfo`. Das Template nutzt es, um zu entscheiden, ob der Loading-Indicator einzublenden ist:

```
<div class="content">
  <div *ngIf="showWaitInfo">
    Please wait ...
  </div>
  [...]
</div>
```

Mit ein paar Styles lässt sich der Loading-Indicator auch gut sichtbar über dem aktuellen Text platzieren.

## Resolver

Unsere FlightEditComponent ermittelt im Live-Cycle-Hook `ngOnInit` die ID des zu ladenden Flugs. Danach könnten wir den Flug anhand dieser ID laden.

Diese zunächst einfache Vorgehensweise führt zu einem Problem: Das Laden des Flugs findet erst statt, nachdem der Routenwechsel abgeschlossen ist. Eine Anwendung, die aufgrund der Routerevents einen Loading-Indicator anzeigt, blendet ihn somit zu früh aus.

Der Grund dafür ist, dass der Router unmittelbar nach Abschluss des Routings das Ereignis `NavigationEnd` auslöst. Zu diesem Zeitpunkt ist das asynchrone Laden des Flugs jedoch noch im Gang. Somit kann der Anwender das Formular nicht sofort nach dem Ausblenden des Indikators nutzen, sondern muss noch ein wenig warten. Auch im Template ist diesem Problem Rechnung zu tragen, um Zugriffe auf die Eigenschaft `flight` zu verhindern, während diese noch `null` bzw. `undefined` ist.

Als Lösung bietet der Router ein Konzept, das sich Resolver nennt. Hierbei handelt es sich in der Regel um Services, die den Typ `Resolve<T>` und dessen Methode `resolve` implementieren. Diese Methode hat die Aufgabe, die von einer Komponente benötigten Werte zu ermitteln. In der Regel stößt sie dazu eine asynchrone Operation an, die einen Promise oder ein Observable zurückliefert. Der Router nimmt diese Objekte entgegen und zögert den Abschluss des Routenwechsels heraus, bis die dahinterstehenden Werte ermittelt wurden. Danach stellt er diese Werte der adressierten Komponente zur Verfügung.

## Vorbereitungen

Für unseren Resolver müssen wir zunächst den `FlightService` erweitern. Wir spendieren ihm eine Methode `findById`:

```
// src/app/flight.service.ts
[...]
export abstract class FlightService {
  [...]
  abstract findById(id: string): Observable<Flight>;
}
```

Der `DummyFlightService` liefert über diese Methode einfach einen hartcodierten Flug zurück:

```
// src/app/dummy-flight.service.ts
[...]
export class DummyFlightService implements FlightService {
  [...]
  findById(id: string): Observable<Flight> {
```

```

        return of({ id: 1, from: 'Frankfurt', to: 'Flagranti',
          date: '2022-01-02T19:00+01:00' });
    }
}

```

Interessanter ist die Implementierung im DefaultFlightService. Diese lädt den gewünschten Flug mit dem HttpClient:

```

// src/app/default-flight.service.ts
[...]
export class DefaultFlightService implements FlightService {
  [...]
  findById(id: string): Observable<Flight> {
    const url = 'http://demo.ANGLARarchitects.io/api/flight';

    const headers = new HttpHeaders()
      .set('Accept', 'application/json');

    const params = new HttpParams()
      .set('id', id);

    return this.http.get<Flight>(url, {headers, params});
  }
}

```

## Resolver erzeugen und verwenden

Auch Resolver lassen sich mittlerweile mit der CLI generieren:

```
ng g resolver flight-booking/flight-edit/flight
```

Da unser FlightResolver einen Flug laden soll, implementiert er das Interface Resolve<Flight> (siehe Beispiel 14-8).

*Beispiel 14-8: Resolver für den zu ladenden Flug*

```

// src/app/flight-booking/flight-edit/flight.resolver.ts

import { Injectable } from '@angular/core';
import {
  Resolve,
  RouterStateSnapshot,
  ActivatedRouteSnapshot
} from '@angular/router';
import { Observable, of } from 'rxjs';
import { Flight } from '../flight';
import { FlightService } from '../flight.service';
import { FlightBookingApiModule } from '../flight-booking-api-module';

@Injectable({
  // providedIn: 'root'
  // Im selben Scope wie den FlugService bereitstellen:
  providedIn: FlightBookingApiModule
})
// Resolve<Flight> nach implements angeben:

```

```

export class FlightResolver implements Resolve<Flight> {

    constructor(private flightService: FlightService) {
    }

    // Resolve<Flight> als Rückgabetyp angeben:
    resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
        Observable<Flight> {

        const id = route.params.id;
        return this.flightService.findById(id);
    }
}

```

Achten Sie bitte darauf, den Resolver im selben Scope wie den FlightService bereitzustellen. Falls Sie das Beispiel zu Lazy Loading in Kapitel 13 umgesetzt haben, ist das das FlightBookingApiModule (nicht das FlightBookingModule!).

Die vom Interface vorgegebene Methode resolve ist für das Laden des Flugs verantwortlich. Wie auch bei Guards werden ihr Informationen über die aktuelle und die zu aktivierende Route übergeben.

Der Resolver lässt sich den FlightService injizieren und ermittelt innerhalb von resolve den Routing-Parameter id. Anschließend fordert resolve beim FlightService den dazugehörigen Flug an und liefert das erhaltene Observable<Flight> zurück.

Die resolve-Methode kann übrigens auch statt eines Observables bzw. eines Promises einen konkreten Wert zurückliefern. Das ist nützlich, wenn sich dieser Wert synchron in Erfahrung bringen lässt.

Die Konfiguration der gewünschten Route referenziert den Resolver über ihre Eigenschaft resolve (siehe Beispiel 14-9).

*Beispiel 14-9: Resolver in Routenkonfiguration hinterlegen*

```

// src/app/flight-booking/flight-booking.routes.ts

[...]
import { FlightResolver } from './flight-edit/flight.resolver';

[...]

export const FLIGHT_BOOKING_ROUTES: Routes = [
    {
        path: '',
        component: FlightBookingComponent,
        children: [
            [...]
            {
                path: 'flight-edit/:id',
                component: FlightEditComponent,
                canDeactivate: [ExitGuard],
                resolve: {
                    flight: FlightResolver

```

```

        }
    }
],
},
];

```

Die Eigenschaft `resolve` verweist auf ein Objekt, dessen Name die Werte wider- spiegelt, die ermittelt werden sollen. Bei den zugewiesenen Werten handelt es sich um die jeweiligen Resolver. Das folgende Listing gibt auf diese Weise bekannt, dass der Wert `flight` mit dem `FlightResolver` zu laden ist.

Um auf die von einem Resolver ermittelten Objekte zuzugreifen, nutzt die adres- sierte Komponente die Eigenschaft `data` der `ActivatedRoute` (siehe Beispiel 14-10).

*Beispiel 14-10: Auf einen Flug zugreifen, der vom Resolver geladen wurde*

```

// src/app/flight-booking/flight-edit/flight-edit.component.ts
[...]
import { Flight } from './flight';

@Component({
  selector: 'app-flight-edit',
  templateUrl: './flight-edit.component.html',
  styleUrls: ['./flight-edit.component.scss']
})
export class FlightEditComponent implements OnInit, CanExit {
  [...]
  ngOnInit(): void {
    [...]
    this.route.data.subscribe(data => {
      this.formGroup.patchValue(data.flight);
    });
  }
}

```

Bei `data` handelt es sich um ein Observable, das die vom Resolver geladenen Ob- jekte liefert. Die geladenen Flüge finden sich in der Eigenschaft `flight`. Dabei han- delt es sich um eine dynamische Eigenschaft, die der Router nach dem Schlüssel im `resolve`-Objekt der Konfiguration (siehe Beispiel 14-9) benennt.

Wenn Sie nun die Anwendung ausführen und die Editieransicht eines Flugs anfor- dern, sollten Sie direkt nach dem Routenwechsel den geladenen Flug angezeigt be- kommen.

Um das Verhalten des Resolvers besser nachvollziehen zu können, können Sie im Resolver eine Verzögerung (`delay`) von ein paar Sekunden einfügen:

```

// src/app/flight-booking/flight-edit/flight.resolver.ts
[...]

```

```

resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
  Observable<Flight> {
  const id = route.params.id;
  return this.flightsService.findById(id).pipe(delay(7000));
}

```

Sie sollten nun sehen, dass der Routenwechsel entsprechend länger dauert und die geladenen Daten trotzdem unmittelbar nach dem Routenwechsel zur Verfügung stehen.

## HttpInterceptoren

Der HttpClient von Angular bietet mit Interceptoren einen Einsprungpunkt. Jeder bereitgestellte Interceptor hat die Möglichkeit, ausgehende HTTP-Anfragen sowie eingehende HTTP-Antworten zu manipulieren. Auf diese Weise lassen sich Anfragen um Authentifizierungsinformationen erweitern, aber auch Caching-Strategien implementieren. Ein weiterer Anwendungsfall ist das Verarbeiten von Datenformaten jenseits von JSON – z.B. XML oder CSV.

Die Idee hinter diesen Interceptoren folgt dem Muster *Chain of Responsibility* (siehe Abbildung 14-2).

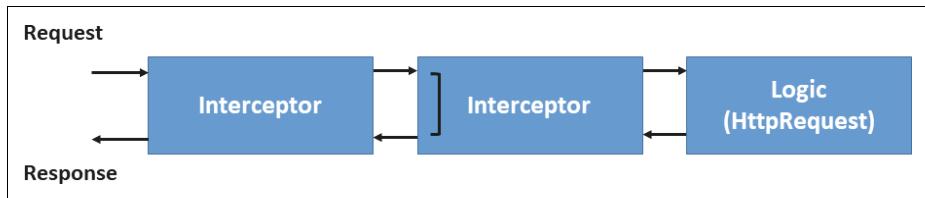


Abbildung 14-2: *Chain of Responsibility*

Dieses Muster sieht vor, dass eine Aktion mit Zusatzlogiken erweitert wird. Letztere werden in Klassen untergebracht, deren Objekte zu einer Kette zusammenge schlossen werden. Am Ende dieser Kette befindet sich die eigentliche Aktion. Jedes dieser Objekte kümmert sich um seine Aufgabe und kann an das nächste Ketten glied delegieren.

Zur Demonstration verwenden wir hier einen Interceptor, der jede ausgehende Anfrage um einen beispielhaften Authorization-Header erweitert. Außerdem leitet er den Benutzer auf die Startseite um, wenn die Antwort auf einen Security-Fehler schließen lässt.

Auch der Interceptor lässt sich mit der CLI generieren:

```
ng g interceptor shared/auth/auth
```

Technisch gesehen, sind HttpInterceptoren lediglich Services, die das Interface `HttpInterceptor` implementieren (siehe Beispiel 14-11).

### Beispiel 14-11: Interceptor für die Authentifizierung

```
// src/app/shared/auth/auth.interceptor.ts

import { Injectable } from '@angular/core';
import {
  HttpRequest,
  HttpHandler,
  HttpEvent,
  HttpInterceptor,
  HttpResponse
} from '@angular/common/http';
import { Observable, throwError } from 'rxjs';
import { Router } from '@angular/router';
import { catchError } from 'rxjs/operators';

@Injectable()
export class AuthInterceptor implements HttpInterceptor {

  constructor(private router: Router) {}

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

    if (req.url.startsWith('http://demo.ANGLUARarchitects.io')) {
      const headers = req.headers.set('Authorization', 'EXAMPLE_ACCESS_TOKEN');
      req = req.clone({ headers });
    }

    return next.handle(req).pipe(
      catchError(resp => this.handleError(resp))
    );
  }

  handleError(resp: HttpErrorResponse): Observable<HttpEvent<any>> {

    if (resp.status === 401 || resp.status === 403) {
      this.router.navigate(['/home', {needsLogin: true}]);
    }
    return throwError(resp);
  }
}
```

Das Interface `HttpInterceptor` gibt lediglich die Methode `intercept` vor, an die Angular die aktuelle Anfrage sowie das nächste Glied der Kette weitergibt. Um die Kontrolle an dieses weiterzugeben, ruft der Interceptor die Methode `next` auf. Das Ergebnis der Methode ist ein Observable mit der HTTP-Antwort. Dieses lässt sich mit den üblichen RxJS-Operatoren bearbeiten.

Bevor der hier gezeigte Interceptor die Anfrage um einen Authorization-Header erweitert, prüft er, ob es sich um eine vertrauenswürdige Web-API handelt. Auf diese Weise stellt er sicher, dass solche vertrauenswürdigen Informationen nicht in die falschen Hände geraten.

Außerdem ist hier zu beachten, dass die API rund um den HttpClient mit Immutables arbeitet. Das bedeutet, dass Interceptoren die headers-Auflistung, aber auch die Anfrage nicht verändern kann. Stattdessen müssen sie diese Objekte klonen und im Rahmen dessen verändern. Bei den Kopfzeilen kümmert sich darum die Methode set. Anstatt die Auflistung zu verändern, liefert sie eine Kopie mit der zusätzlichen Kopfzeile. Bei der Anfrage kommt für die gleiche Aufgabe die Methode clone zum Einsatz.

Damit Angular den Interceptor verwendet, muss die Anwendung ihn für das Token `HTTP_INTERCEPTORS` registrieren. Das kann zum Beispiel im AppModule erfolgen (siehe Beispiel 14-12).

*Beispiel 14-12: HttpInterceptor im AppModule registrieren*

```
// src/app/app.module.ts

[...]

// Hinzufügen:
import { HTTP_INTERCEPTORS } from '@angular/common/http';
import { AuthInterceptor } from './shared/auth/auth.interceptor';

@NgModule({
  ...
  providers: [
    {
      provide: HTTP_INTERCEPTORS,
      useClass: AuthInterceptor,
      multi: true
    },
    ...
  ]
})
export class AppModule { }
```

Die Option `multi` ist hier auf `true` zu setzen, um Angular darüber zu informieren, dass es mehrere Interceptoren geben kann. All diese bilden die oben erwähnte Kette. Die Reihenfolge der Registrierungen repräsentieren die Reihenfolge der Glieder innerhalb der Kette.

Um zu prüfen, ob das Access-Token übersendet wird, können Sie in Chrome das Registerblatt *Network* in den Dev-Tools nutzen (siehe Abbildung 14-3).

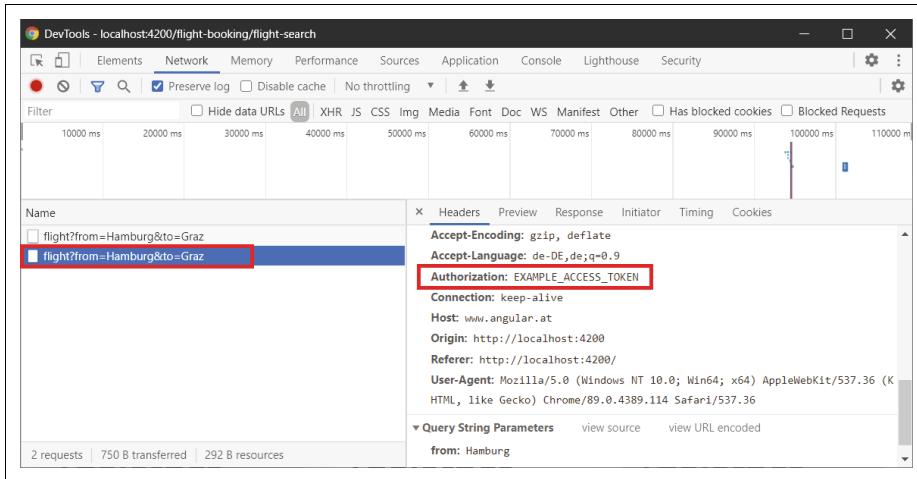


Abbildung 14-3: Access-Token in HTTP-Anfrage

Das Verhalten zur Fehlerbehandlung können Sie testen, indem Sie zwischenzeitlich mit dem `DefaultFlightService` auf die folgende URL zugreifen:

<http://demo.ANGLARarchitects.io/api/error?code=401>

Diese zu Testzwecken eingerichtete API liefert einen Fehler mit dem übergebenen Statuscode zurück.

## Zusammenfassung

Angular bietet einige Möglichkeiten zum Umsetzen von Querschnittsfunktionen. Mit den Guards kann die Anwendung zum Beispiel dem Router mitteilen, welche Routenwechsel erlaubt sind. Resolver kommen hingegen zum Einsatz, um Routenwechsel zu verzögern, bis benötigte Daten vorliegen.

Ein weiteres mächtiges Konzept für Querschnittsfunktionen sind HttpInterceptoren. Sie erlauben es, ausgehende HTTP-Anfragen und ankommende HTTP-Antworten global zu betrachten bzw. zu manipulieren. Auf diese Weise lassen sich zum Beispiel sämtliche Anfragen um Security-Kopfzeilen erweitern, Fehler global behandeln oder Datenformate jenseits von JSON parsen.

# Authentifizierung und Autorisierung

Die wenigsten Geschäftsanwendungen kommen ohne Authentifizierung aus. Dieses Kapitel geht auf zwei Implementierungsvarianten ein. Wir starten mit der klassischen Cookie-basierten Authentifizierung und werfen danach einen Blick auf die tokenbasierte Authentifizierung mit *OAuth 2* und *OpenID Connect* (OIDC). Im Zuge dessen binden wir einen existierenden Identity Provider an.

## Cookie-basierte Security

Die Nutzung von Cookies ist nicht nur einfach, sondern gilt dank zahlreicher Weiterentwicklungen auch als sicher. Für Letzteres sorgen Attribute, die in den letzten Jahren definiert wurden (siehe Tabelle 15-1).

Tabelle 15-1: Attribute zum Absichern von Cookies

Attribut	Bedeutung
HttpOnly	Das Cookie kann nicht via JavaScript gelesen werden. Schadcode kann es also nicht direkt entwenden.
Secure	Das Cookie darf nur via HTTPS verwendet werden.
SameSite	Schränkt die Nutzung von Cookies über Cross-Origin-Requests ein. Folgen Sie beispielsweise einem Link, der von einer anderen Website auf Ihre verweist, wird ein Same-Site-Cookie nicht in die Anfrage inkludiert.

Auf der Clientseite haben Sie keine Möglichkeit, diese Attribute zu beeinflussen. Diese definiert der Server beim Ausstellen der Cookies. Aus der Perspektive des Clients ist die Nutzung von Cookies somit recht simpel:

1. Der Client ruft eine API auf.
2. Die API authentifiziert den Benutzer und stellt ein Cookie aus.
3. Der Client ruft weitere APIs auf. Dabei inkludiert der Browser ohne weiteres Zutun die Cookies. Diese teilen der API mit, um welchen Benutzer es sich handelt.

Zur Authentifizierung des Benutzer im zweiten Schritt existieren verschiedene Optionen. Die Anwendung kann beispielsweise Benutzername und Passwort übersen-

den. Alternativ dazu kann das Backend in Schritt 2 auch an eine bestehende Identity-Lösung wie Active Directory delegieren und ein signiertes Security-Token mit Informationen zum Benutzer erhalten. Wir gehen weiter unten darauf etwas genauer ein.

## Cookies und XSRF

Bei *Cross Site Request Forgery* (CSRF oder XSRF) handelt es sich um eine Attacke, bei der ein Angreifer einen authentifizierten Benutzer dazu bringt, ohne sein Wissen eine Aktion bei einer Webanwendung anzustoßen. Im Wesentlichen läuft diese Attacke wie folgt ab:

1. Der Benutzer meldet sich bei Ihrer Webanwendung an und erhält ein Cookie.
2. Der Benutzer begibt sich auf eine von einem Angreifer kontrollierte Seite.
3. Diese Seite greift auf Ihre Webanwendung zu. Beispielsweise bietet sie ein Formular an, das seine Daten an Ihre Website übersendet, oder einen Hyperlink, der Ihre Seite mit bestimmten Parametern aufruft. Daneben existieren noch weitere Möglichkeiten.
4. Der Benutzer sendet das Formular ab bzw. folgt dem Hyperlink. Da der Browser bei Zugriffen auf Ihr Backend das Cookie hinzufügt, stoßen dort diese Formulare bzw. Hyperlinks eine Aktion in Ihrem Namen an.

Die im letzten Abschnitt erwähnten Same-Site-Cookies schränken diese Attacke schon erheblich ein. Eine weitere Möglichkeit zur Eindämmung ist der Einsatz von XSRF-Tokens (nicht zu verwechseln mit den Security-Tokens im nächsten Abschnitt). Dabei handelt es sich um eine zufällige Zeichenkette, die Ihre Angular-Anwendung vom Backend beim Start bekommt. Diese Zeichenkette muss die Angular-Anwendung in jede HTTP-Anfrage inkludieren. Somit beweist sie, dass die Anfrage von ihr und nicht von einer von einem Angreifer kontrollierten Seite stammt.

Diesen Schutzmechanismus implementiert der HttpClient von Haus aus. Er erwartet beim Aufruf der Angular-Anwendung das Token in einem Cookie mit dem Namen XSRF-TOKEN. Der HttpClient merkt sich seinen Wert und sendet ihn im Rahmen jeder einzelnen Anfrage über den Header X-XSRF-TOKEN zurück an den Server.

Sie müssen somit lediglich dafür sorgen, dass der Server zum einen das Cookie XSRF-TOKEN ausstellt und zum anderen bei jedem weiteren Aufruf den Header X-XSRF-TOKEN prüft. Wichtig ist dabei, dass der Server nach jeder Anmeldung für den jeweiligen Benutzer ein neues Token, das Angreifer nicht vorausahnen können, ausstellt.

Wollen Sie die Namen dieses Cookies oder Headers ändern, müssen Sie lediglich das `HttpClientXsrfModule` in Ihr `AppModule` importieren:

```
// Entnommen aus der offiziellen Angular-Dokumentation:  
imports: [  
  HttpClientModule,
```

```
HttpClientXsrfModule.withOptions({
  cookieName: 'My-Xsrf-Cookie',
  headerName: 'My-Xsrf-Header',
}),
],
```

## Tokenbasierte Security

Häufig müssen bestehende Identity-Lösungen wie Active Directory oder LDAP-Systeme integriert werden, um Single-Sign-on zu ermöglichen. In modernen Webanwendungen muss der Client auch das Recht erhalten, im Namen des angemeldeten Benutzers auf Services zuzugreifen. All diese Anforderungen lassen sich elegant mit Security-Tokens lösen.

In diesem Kapitel zeigen wir Ihnen, wie tokenbasierte Sicherheit in einer Angular-Anwendung genutzt werden kann. Dazu kommen die populären Standards *OAuth 2* und *OpenID Connect* zum Einsatz, um die in diesem Buch verwendete Demoanwendung in Hinblick auf Authentifizierung und Autorisierung zu erweitern.



Falls Sie sämtliche API-Aufrufe über ein einziges Backend bedienen können, spricht nichts dagegen, die tokenbasierte Security hinter diesem Backend zu verstecken. Das Backend stellt in diesem Fall nach einer erfolgreichen Authentifizierung bei der Identity-Lösung ein Cookie aus. Aus Sicht der Angular-Anwendung gestaltet sich somit der Ablauf wie in den vorangegangenen Abschnitten beschrieben.

## OAuth 2 und OpenID Connect

Wer sich heutzutage mit tokenbasierter Sicherheit beschäftigt, kommt wohl kaum an den beiden populären Standards *OAuth 2* (<https://oauth.net/2/>) und *OpenID Connect* (<http://openid.net/connect/>) vorbei. Sie beschreiben unter anderem, wie sich ein Benutzer bei einem verteilten System anmelden kann und wie ein Client das Recht erhält, im Namen des Benutzers Services zu konsumieren. Dazu kommt, dass diese Standards direkt auf HTTPS aufsetzen und sich somit wunderbar für leichtgewichtige Web-APIs eignen.



Sowohl OAuth 2 als auch OpenID Connect müssen über HTTPS verwendet werden, um sicher zu sein. In der hier verwendeten Demoanwendung nehmen wir jedoch aus Gründen der Vereinfachung davon Abstand.

## OAuth 2

Die erste Version von OAuth wurde 2006 von Twitter und Ma.gnolia entwickelt. Das Ziel war es, Benutzern die Möglichkeit zu geben, einen Teil ihrer Rechte an einen Client weiterzugeben, ohne das eigene Passwort mit ihm zu teilen. Somit kön-

nen zum Beispiel Anwendungen das Recht erhalten, im Namen des Benutzers Services aufzurufen.

Mittlerweile wird OAuth bzw. dessen Nachfolger OAuth 2 von Größen wie Google, Facebook, Flickr, Microsoft, Salesforce.com oder Yahoo! eingesetzt. Dabei fällt auf, dass es zunehmend nicht nur zum Delegieren von Rechten (Autorsierung), sondern auch für Single-Sign-on-Szenarien (Authentifizierung) eingesetzt wird. So können sich Benutzer zum Beispiel mit ihrem Google-Konto auch bei anderen Weblösungen anmelden. In diesem Fall erhält die betroffene Weblösung das Recht, auf die Profildaten des angemeldeten Google-Benutzers zuzugreifen. Auch andere der zuvor aufgeführten Unternehmen bieten diese Möglichkeit.

Abbildung 15-1 verdeutlicht die Funktionsweise von OAuth 2 aus der Vogelperspektive.

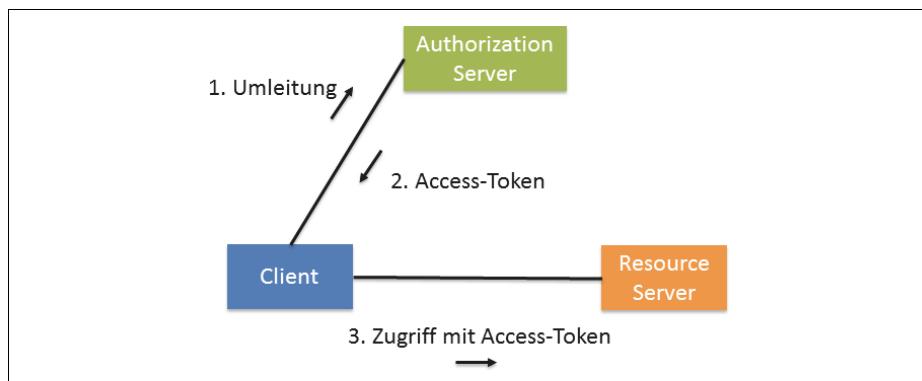


Abbildung 15-1: Die Funktionsweise von OAuth 2 aus der Vogelperspektive

Der Client leitet den Benutzer zur Anmeldung zu einem sogenannten *Authorization Server* weiter. Diese Instanz hat Zugriff auf zentrale Benutzerkonten. Hat sich der Benutzer dort angemeldet, erhält der Client ein sogenanntes Access-Token, das ihm im Namen des Benutzers Zugriff auf Services im Backend – sogenannte *Resource Server* – gibt.

Ein Access-Token informiert den Resource Server unter anderem über den entsprechenden Benutzer sowie über die Rechte, die der Client im Namen des Benutzers wahrnehmen darf. Zusätzlich finden sich im Token meist auch Metadaten wie der Aussteller, das Ausstellungsdatum oder die Gültigkeitsdauer.

Diese vom Prinzip her einfache Vorgehensweise hat mehrere Vorteile:

- Jeder Benutzer kann ein zentrales Benutzerkonto für verschiedene Clients und Services nutzen.
- Da die Anmeldung beim Authorization Server erfolgt, bekommt der Client das Passwort nicht in die Hand.
- Die Authentifizierung ist vom Client entkoppelt und lässt sich somit in bestehende Identity-Lösungen integrieren.

- Tokens erhöhen die Flexibilität. Beispielsweise könnte ein Service das Token an einen weiteren Service weiterreichen, um zu beweisen, dass er im Namen des Benutzers agiert. Zum Zugriff auf andere Sicherheitsdomänen kann der Service das Token auch gegen eines für diese Domäne tauschen.
- Die Lösung kommt ohne Cookies aus. Somit kann der Client ebenfalls auf Services zugreifen, die auf anderen Servern laufen (bzw. einen anderen Ursprung [Origin] haben). Zusätzlich schränkt der Verzicht auf Cookies bestimmte Angriffe ein.

Das Format des Access-Tokens sowie die Maßnahmen, die der Resource Server zum Validieren des Tokens unternimmt, sind Implementierungsdetails, die OAuth 2 nicht näher beschreibt. Häufig kommen digitale Signaturen zum Einsatz, damit der Resource Server einfach prüfen kann, ob das Token von einem vertrauenswürdigen Authorization Server stammt. Alternativ dazu könnte das Token auch nur aus einer nicht vorhersehbaren ID bestehen, mit der der Resource Server sich nochmals an den Authorization Server wendet.

## Benutzer mit OpenID Connect authentifizieren

Als Ergänzung zu OAuth 2 definiert OpenID Connect (OIDC) unter anderem, wie der Client Informationen über den Benutzer bekommen kann. Diesen Aspekt deckt OAuth 2 nicht ab, und selbst das ausgestellte Token muss für den Client nicht lesbar sein. Deswegen spezifiziert OIDC unter anderem ein sogenanntes ID-Token, das der Client zusätzlich zum Access-Token erhalten kann. Während das Access-Token zum Zugriff auf das Backend bestimmt ist, kann der Client dem ID-Token direkt Informationen über den Benutzer entnehmen (siehe Abbildung 15-2).

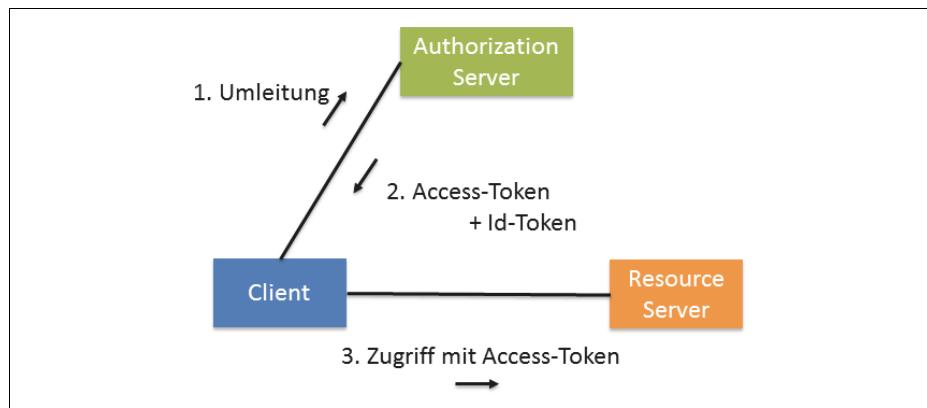


Abbildung 15-2: Die Funktionsweise von OpenID Connect aus Vogelperspektive

Im Gegensatz zu Access-Tokens bei OAuth 2 ist der Aufbau von ID-Tokens vorgegeben. Es handelt sich dabei immer um ein **JSON Web Token (JWT)**, das signiert und/oder verschlüsselt sein kann. Zusätzlich definiert OIDC einen Userinfo-Endpunkt. Dabei handelt es sich um einen HTTP-Service, der dem Client weitere In-

formationen zum aktuellen Benutzer preisgibt, sofern dieser das erhaltene Access-Token vorweisen kann. Bei diesen Informationen könnte es sich um die postalische Adresse oder um das Profilbild des Benutzers handeln. Welche Informationen bereits im ID-Token vorkommen und welche über den userinfo-Endpunkt anzufordern sind, ist bei den meisten Identity-Lösungen eine Frage der Konfiguration.

## JSON Web Token

Ein JSON Web Token (JWT) beinhaltet unter anderem ein JSON-basiertes Objekt mit Claims. Dabei handelt es sich um Name/Wert-Paare, die ein Subjekt beschreiben, zum Beispiel einen Benutzer. Daneben existieren Claims, die Informationen über das Token selbst liefern, darunter den Zeitraum, in dem das Token gültig ist, oder die Audience des Tokens. Dieses Claims-Set kann der Aussteller signieren und/oder verschlüsseln. Nachfolgend findet sich ein Beispiel für ein signiertes JWT. Es besteht aus drei BASE64-codierten Abschnitten, die durch einen Punkt zu trennen sind. Zur besseren Lesbarkeit haben wir in diesem Listing Zeilenumbrüche eingefügt und den zweiten und dritten Teil unter Verwendung von Auslassungszeichen abgekürzt:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9  
.  
eyJuYmYiOjEz[...]BlbmlkIn0  
.  
Nt5pBRqGvDFn[...]1205awFjw
```

Der erste Teil beinhaltet ein JSON-Objekt, das den Header des JWT repräsentiert. Der zweite Teil enthält das Claims-Set und der dritte Teil die Signatur. Der Header des betrachteten JWT gestaltet sich wie folgt:

```
{"typ": "JWT", "alg": "RS256"}
```

Interessant ist hierbei die Eigenschaft `alg`, die den Algorithmus widerspiegelt, der zur Erstellung der Signatur verwendet wurde. RS256 bedeutet, dass der Aussteller aus dem zu signierenden Claims-Set mit SHA-256 einen Hash-Wert errechnet und für diesen anschließend mit RSA eine digitale Signatur erstellt hat. Da es sich bei RSA um ein asymmetrisches Verfahren handelt, hat der Aussteller zum Signieren einen privaten Schlüssel eingesetzt. Ob die Signatur korrekt ist, kann nun jeder mit einem öffentlichen Schlüssel prüfen.

Das folgende Listing zeigt das Claims-Set, das sich im zweiten Teil des JWT befindet:

```
{  
  "nbf":1388357979,  
  "exp":1388444379,  
  "aud":[  
    "http://service",  
    "http://partner-authsvc",  
    "http://myClient"],  
  "iss":"http://authsvc",  
  "sub":"3ca4ccc8",  
  "name":"Manfred Steyer",
```

```
    "role": "Manager",
    "company": "ACME"
}
```

Die Claims `nbf` (*not before*) und `exp` (*expiration time*), die einen UNIX-Timestamp (Sekunden seit dem 1.1.1970, 0 Uhr GMT) beinhalten, geben die Zeitspanne an, in der das Token gültig ist. Die Audience des Tokens findet sich im Claim `aud`. Es handelt sich dabei um ein JSON-Array mit den einzelnen Parteien, für die das Token ausgestellt wurde. Wird ein JWT nur für eine Partei ausgestellt, kann dieses Claim auch nur aus einem String mit dem Bezeichner dieser Partei bestehen. Der Aussteller des Tokens findet sich im Claim `iss` (*issuer*) wieder, und das Claim `sub` repräsentiert das Subjekt, das durch das vorliegende Claims-Set beschrieben wird. Im betrachteten Fall handelt es sich hierbei um eine Benutzer-ID. Die restlichen Claims beinhalten den Namen (`name`), die Firma (`company`) und die Rolle (`role`) des beschriebenen Benutzers.

Während sich Aussteller und Konsumenten von Claims bilateral auf die zu verwendenden Claim-Namen einigen können, ist es sinnvoll, zu prüfen, ob es für den gewünschten Zweck bereits offiziell definierte Namen gibt, um Kollisionen sowie Missverständnisse zu vermeiden. Eine gute erste Anlaufstelle dafür ist die OpenID-Connect-Spezifikation (<http://openid.net/connect>). Darüber hinaus kann der Aussteller auch öffentliche Bezeichner, zum Beispiel URLs, als Namen für Claims heranziehen.

## OAuth-2- und OIDC-Flows

Für verschiedene Anwendungsfälle wurden im Laufe der Zeit verschiedene Flows entwickelt. Diese geben die Nachrichten vor, die ausgetauscht werden müssen, damit der Client zum gewünschten Access- bzw. Identity-Token kommt.

Für Single Page Applications wurde ursprünglich der sogenannte Implicit Flow definiert. Dieser gleicht der Übersicht im letzten Abschnitt. Mittlerweile wird jedoch für Single Page Applications der *Authorization Code Flow* im Zusammenspiel mit dem Sicherheitsmechanismus *Proof Key for Code Exchange* (PKCE) empfohlen, und der Implicit Flow gilt mit OAuth 2.1 als veraltet (deprecated).

Eine detaillierte Diskussion der Unterschiede zwischen diesen Flows würde den Umfang dieses Buchs sprengen. Fürs Erste müssen Sie lediglich folgendes wissen:

- Für neue Anwendungen sollten Sie Bibliotheken und APIs nutzen, die Authorization Code Flow und PKCE unterstützen.
- Bestehende Lösungen, die gemäß der ursprünglichen Empfehlung mit dem Implicit Flow entwickelt wurden, sind nach wie vor sicher, sofern die vorherrschenden Best Practices eingehalten wurden.

Wer mehr dazu wissen möchte, wird in unserem Vortrag unter <https://www.youtube.com/watch?v=p80Cvw21X14> sowie in den Spezifikationen zu OAuth 2 und OIDC fündig.

# OAuth 2 und OIDC mit Angular nutzen

Um den Einsatz von OAuth 2/OIDC in einer Angular-Anwendung zu demonstrieren, nutzen wir nun unsere Bibliothek angular-oauth2-oidc (<https://www.npmjs.com/package/angular-oauth2-oidc>):

```
npm install angular-oauth2-oidc --save
```

Um die Bibliothek nach dem Herunterladen der Angular-Anwendung bekannt zu machen, müssen Sie das OAuthModule ins AppModule importieren (siehe Beispiel 15-1).

*Beispiel 15-1: OAuthModule referenzieren*

```
// app.module.ts

[...]

// Hinzufügen:
import { OAuthModule } from 'angular-oauth2-oidc';

@NgModule({
  imports: [
    HttpClientModule,
    // Hinzufügen:
    OAuthModule.forRoot(),
    [...]
  ],
  ...
})
export class AppModule {
```

Danach müssen Sie die Bibliothek konfigurieren. Wir legen dazu eine Datei *auth.config.ts* an (siehe Beispiel 15-2).

*Beispiel 15-2: Konfiguration für angular-oauth2-oidc*

```
// src/app/auth.config.ts

import { AuthConfig } from 'angular-oauth2-oidc';

export const authConfig: AuthConfig = {
  issuer: 'https://idsrv4.azurewebsites.net',
  redirectUri: window.location.origin + '/index.html',
  clientId: 'spa',
  // Use Code Flow
  responseType: 'code',
  scope: 'openid profile email api',
};
```

Zu diesen Informationen gehören die URL des Authorization Server (issuer), aber auch die ID des Angular-Clients sowie dessen URL. Aus Sicherheitsgründen müs-

sen diese beiden Informationen im Vorfeld beim Authorization Server registriert werden. Auf diese Weise stellt er sicher, dass tatsächlich der Client mit der angegebenen ID und somit jener Client, für den sich der Benutzer anmeldet, die Tokens erhält.

Der Scope repräsentiert die einzelnen Berechtigungen, die der Client im Namen des Benutzers durchführen möchte. Die ersten drei hier definierten Werte stammen aus der Welt von OpenID Connect. Sie erlauben Zugriff auf die Benutzer-ID (`openid`), auf Profilinformationen wie Vorname und Nachname (`profile`) und auf die E-Mail-Adresse (`email`) des Benutzers. Der vierte Scope (`api`), der im Authorization Server zu definieren ist, ist Usecase-spezifisch und ermöglicht Zugriff auf die Web-API.

Weitere Eckdaten bezieht der Client über das Discovery Document, das der Authorization Server bereitstellt. Dabei handelt es sich um ein durch OpenID Connect definiertes JSON-Dokument, das unter anderem die einzelnen Endpunkte zum Anfordern von Tokens oder Benutzerinformationen widerspiegelt. Per definitionem findet sich dieses unter der URL, die sich ergibt, wenn man an die URL des Authorization Server die Segmente `.well-known/openid-configuration` anhängt. In Fällen, in denen der Authorization Server Ihrer Wahl kein solches Dokument anbietet, nimmt die Bibliothek die einzelnen Einstellungen auch über Eigenschaften entgegen. Informationen dazu finden sich in der Dokumentation (<https://www.npmjs.com/package/angular-oauth2-oidc>).

Als Nächstes übergeben wir beim Programmstart die Konfiguration an die Bibliothek. Dazu injizieren wir den `OAuthService` in die `AppComponent` (siehe Beispiel 15-3).

*Beispiel 15-3: Konfiguration an die Bibliothek übergeben*

```
// src/app/app.component.ts

[...]

// Hinzufügen:
import { OAuthService } from 'angular-oauth2-oidc';
import { authConfig } from './auth.config';

@Component({ ... })
export class AppComponent implements OnInit {

    [...]

    constructor(
        private oauthService: OAuthService,
        [...]) {
            oauthService.configure(authConfig);
            oauthService.loadDiscoveryDocumentAndTryLogin();
    }

    [...]
}
```

Die Methode `configure` nimmt die Konfiguration entgegen. Die Methode `loadDiscoveryDocumentAndTryLogin` lädt weitere Konfigurationsinformationen vom Authorization Server und startet die Bibliothek.

Nun müssen wir noch unseren AuthService aktualisieren, sodass dieser den Benutzer via OAuth 2/OIDC an- und abmeldet (siehe Beispiel 15-4).

*Beispiel 15-4: Der AuthService verwendet nun OAuth 2/OIDC.*

```
// src/app/shared/auth/auth.service.ts

import { Injectable } from '@angular/core';
import { OAuthService } from 'angular-oauth2-oidc';

@Injectable()
export class AuthService {

  get userName(): string | null {
    const claims = this.oauthService.getIdentityClaims() as any;
    if (!claims) {
      return null;
    }
    return claims.given_name;
  }

  constructor(private oauthService: OAuthService) { }

  login(userName: string): void {
    this.oauthService.initLoginFlow();
  }

  logout(): void {
    this.oauthService.logOut();
  }
}
```

Die Methode `getIdentityClaims` liefert die Claims, mit denen das Identity-Token den aktuellen Benutzer beschreibt. OIDC standardisiert ein paar Claims, wie z.B. den Vornamen des Benutzers (`given_name`).

Der OAuthService bietet noch ein paar weitere nützliche Methoden an:

`hasValidAccessToken`

Liefert true, wenn ein gültiges Access-Token existiert.

`hasValidIdToken`

Liefert true, wenn ein gültiges Identity-Token existiert.

`getAccessToken`

Liefert das aktuelle Access-Token als String.

`getIdToken`

Liefert das aktuelle Identity-Token als String.

Um nun mit dem Access-Token auf das Backend zuzugreifen, müssen Sie es lediglich in die HTTP-Header aufnehmen (siehe Beispiel 15-5).

*Beispiel 15-5: Access-Token manuell per Header übersenden*

```
// src/app/default-flight.service.ts

[...]

// Hinzufügen:
import { OAuthService } from 'angular-oauth2-oidc';

@Injectable()
export class DefaultFlightService implements FlightService {

  constructor(
    // Hinzufügen:
    private oauthService: OAuthService,
    private http: HttpClient) { }

  [...]

  find(from: string, to: string): Observable<Flight[]> {
    const url = 'http://demo.ANGULARarchitects.io/api/flight';

    const headers = new HttpHeaders()
      .set('Accept', 'application/json')
    // Hinzufügen:
    .set('Authorization', this.oauthService.authorizationHeader());

    const params = new HttpParams()
      .set('from', from)
      .set('to', to);

    return this.http.get<Flight[]>(url, {headers, params});
  }
}
```

Damit Sie das nicht bei jedem Aufruf separat machen müssen, bietet sich der Einsatz eines *HttpInterceptors* an. Um zyklische Abhängigkeiten zu vermeiden, sollten Sie in den Interceptor lediglich den sogenannten *OAuthStorage* anstelle des *OAuth Service* importieren. Eine entsprechende Umsetzung, die auf dem *HttpInterceptor* aus Kapitel 14 aufsetzt, finden Sie in Beispiel 15-6.

*Beispiel 15-6: Der HttpInterceptor übersendet nun das Access-Token.*

```
// src/app/shared/auth/auth.interceptor.ts

[...]

// Hinzufügen:
import { OAuthStorage } from 'angular-oauth2-oidc';

@Injectable()
export class AuthInterceptor implements HttpInterceptor {

  constructor(
```

```

// Hinzufügen:
private storage: OAuthStorage,
private router: Router) {
}

intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

  if (req.url.startsWith('http://demo.ANGLARArchitects.io')) {
    // Aktualisieren:
    const headers =
      req.headers.set('Authorization', 'Bearer ' + this.storage.getItem('access_token'));
    req = req.clone({ headers });
  }

  return next.handle(req).pipe(
    [...]
  );
}
[...]
}

```

In vielen Fällen ist das manuelle Umsetzen eines solchen Interceptors jedoch gar nicht notwendig, weil die Bibliothek `angular-oauth2-oidc` bereits einen solchen enthält. Um ihn zu aktivieren, müssen Sie lediglich beim Importieren des Moduls angeben, an welche URLs das Access-Token zu übersenden ist:

```

// app.module.ts

[...]
OAuthModule.forRoot({
  resourceServer: {
    allowedUrls: ['http://demo.ANGLARArchitects.io/api'],
    sendAccessToken: true
  }
})
[...]

```

Die Bibliothek hat noch ein wenig mehr zu bieten. Beispielsweise kann sie das erhaltene Token bei Bedarf aktualisieren. Details dazu finden Sie in der unter <https://www.npmjs.com/package/angular-oauth2-oidc> verlinkten Dokumentation.

## Zusammenfassung

Cookies sind eine einfache und sichere Möglichkeit zum Authentifizieren und Autorisieren der Benutzer.

Für das Anbinden bestehender Identity-Lösungen wie Active Directory bieten sich hingegen die Standards OAuth 2 und OpenID Connect an. Sie definieren, wie eine Anwendung zu Security-Tokens kommt. Sogenannte ID-Tokens beschreiben für die Anwendung, welcher Benutzer gerade vor dem Bildschirm sitzt, und Access-Tokens geben dem Client Zugriff auf das Backend. Durch den Einsatz dieser Standards können Sie flexible Security-Szenarien für moderne Single Page Applications umsetzen und externe Identity-Lösungen einbinden.

# Internationalisierung

Es ist nicht unüblich, Anwendungen an verschiedene Regionen und Sprachen anzupassen. Dies gilt es jedoch bereits bei der Umsetzung zu berücksichtigen, z. B. indem Texte austauschbar gestaltet werden. Dasselbe gilt auch für Formate, für Datumswerte und Zahlen. Genau das ist das Ziel der Internationalisierung, auch kurz *I18N* genannt. Die Buchstaben *I* und *N* stehen dabei für den ersten und letzten Buchstaben der englischen Entsprechung *Internationalization* und 18 für die Anzahl an Buchstaben dazwischen.

Die auf die I18N folgende Anpassung für einzelne Sprachen und/oder Regionen nennt man übrigens auch Lokalisierung (*Localization* bzw. *L10N*). Bei der erwähnten Kombination aus Region und Sprache ist von *Locale* die Rede. Manche deutsche Werke sprechen hierbei auch von Kulturen. Ein Beispiel dafür ist *de-DE* für Deutsch, wie es in Deutschland gesprochen wird (Bundesdeutsch). Das Locale *de-AT* würde sich hingegen auf österreichisches Deutsch beziehen. Die sogenannten generischen Locales gehen nicht auf regionale Unterschiede ein. Ein Beispiel dafür wäre *de* für Deutsch ohne Bezug zu einer Region.

In diesem Kapitel zeigen wir Ihnen, wie die I18N-Lösung von Angular funktioniert. Durch die Nutzung von Compilertools unterscheidet sich der Ansatz deutlich von den meisten Frameworks. Das führt auch, neben den Vorteilen, zu Einschränkungen, die wir noch näher beschreiben werden. Als Alternative gehen wir im zweiten Abschnitt auf *ngx-translate* ein – eine Bibliothek aus der Community, die bereits große Verbreitung findet und mehr dem Ansatz der üblichen Werkzeuge in diesem Segment entspricht.

## I18N mit dem Angular-Compiler

Die in Angular integrierte Lösung für I18N basiert auf dem Angular-Compiler. Dieser ist in der Lage, Texte aus den Templates zu extrahieren und in XML- bzw. JSON-Dateien auszulagern. Nachdem die Dateien übersetzt wurden, integriert der Compiler sie in die generierten Bundles.

In diesem Abschnitt gehen wir auf dieses Verfahren und die damit verbundenen Konsequenzen ein.

## Überblick

Abbildung 16-1 zeigt den prinzipiellen Ablauf bei Nutzung der compilerbasierten I18N.

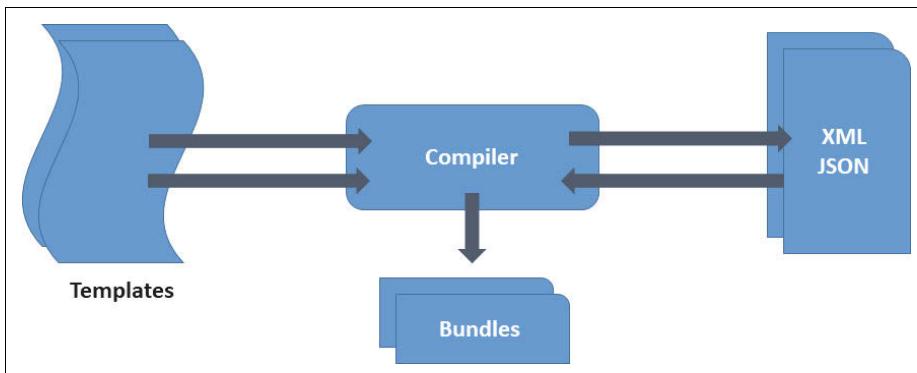


Abbildung 16-1: Der Angular-Compiler extrahiert Texte aus den Templates.

Die vom Compiler extrahierten Dateien unterliegen einem von mehreren möglichen Standards, die sich der Unterstützung von Softwareprodukten in Übersetzungsstudios erfreuen. Neben XLIFF (*XML Localization Interchange File Format*) in Version 1 und 2 und XMB (*XML Message Bundles*) unterstützt der Compiler neuerdings auch JSON-basierte Formate wie zum Beispiel ARB (*Application Resource Bundle*).

Nach dem Übersetzen dieser Dateien integriert der Compiler deren Texte in die erzeugten Bundles. Das bedeutet, dass es pro Sprache ein eigenes Set an Bundles gibt. Der große Vorteil dabei ist, dass die Anwendung zur Laufzeit keinen Aufwand hat, um die Texte zu laden oder darzustellen – sie sind integraler Bestandteil der Bundles.

Auf der anderen Seite müssen Sie jedoch auch mehrere Sprachversionen bereitstellen. Zum Wechsel der Sprache sind Hyperlinks, die auf die jeweils anderen Sprachversionen verweisen, einzurichten. Bei jedem Sprachwechsel lädt der Browser somit eine neue Anwendung, und der Zustand der zuvor geladenen Sprachversion geht verloren. Das ist der Preis, der für die gute Laufzeitperformance zu zahlen ist.



Ursprünglich hat die Angular CLI tatsächlich pro Sprache den gesamten Build-Vorgang ausgeführt. Es liegt auf der Hand, dass sehr langsame Builds die Folge waren.

Seit Version 9 geht die CLI hier schlauer vor: Sie führt nur einen einzigen Build-Vorgang aus und integriert dabei Platzhalter für alle zu tauschenden Texte. Anschließend kopiert sie das so erhaltene Bundle-Set pro Sprache und tauscht die Platzhalter gegen deren Texte aus. Somit ergibt sich nun eine weitaus bessere Build-Performance.

Nachdem wir nun die prinzipielle Funktionsweise und die Konsequenzen der Compiler-I18N geklärt haben, betrachten wir in den nächsten Abschnitten die konkreten Schritte, um diese zu nutzen.

## @angular/localize installieren

Die Compiler-I18N ist zwar ein integraler Bestandteil des Angular-Compilers, allerdings benötigen wir auch Werkzeuge für die CLI und die Nutzung zur Laufzeit. Diese stellt das Angular-Team über das Paket @angular/localize zur Verfügung. Sie können es mit ng add installieren:

```
ng add @angular/localize
```

## Texte markieren

Damit der Compiler weiß, welche Texte zu extrahieren sind, müssen wir diese in den Templates mit dem Attribut i18n markieren:

```
<!-- src/app/flight-search/flight-search.component.html -->  
  
<h1 i18n="Bedeutung|Beschreibung@@flightSearch-title">Flight Search</h1>
```

Der an i18n zugewiesene Wert kann aus drei Teilen bestehen, wobei alle drei optional sind. Die Zeichen | sowie @@ fungieren als Trennzeichen. Die Bedeutung und die Beschreibung geben dem Übersetzungsstudio die nötigen Kontextinformationen. Diese sind gerade dann notwendig, wenn für ein Wort mehrere mögliche Übersetzungen vorliegen. Der Wert nach @@ stellt eine ID dar. Der Compiler nutzt sie, um sich später daran zu erinnern, wo welcher Text zu platzieren ist. Geben Sie keine ID an, generiert der Compiler eine. Das ist jedoch nicht ideal, zumal sich die generierte ID nach Änderungen am Template ändern kann und der Compiler somit Probleme beim Zuordnen der Übersetzungen hat. Insofern ist es gute Praxis, die ID selbst zuzuweisen. Da alle drei Angaben optional sind, ergeben sich verschiedene mögliche Kombinationen:

```
<h1 i18n="Beschreibung">Flight Search</h1>  
<h1 i18n="Beschreibung@@flightSearch-title">Flight Search</h1>  
<h1 i18n="@@flightSearch-title">Flight Search</h1>  
<h1 i18n>Flight Search</h1>
```

Es ist übrigens auch möglich, Datenbindungen in solchen Texten zu platzieren (siehe Beispiel 16-1).

*Beispiel 16-1: Zu übersetzende Texte können auch Datenbindungen beinhalten.*

```
<!-- src/app/flight-search/flight-search.component.html -->  
  
[...]  
  
<div class="form-group" *ngIf="flights$ | async as flights">  
  
[...]  
  
<div i18n="@@flightSearch-flightsFound">{{flights.length}} flights found!</div>  
  
[...]  
  
</div>
```



Darüber hinaus können Sie einzelne Attribute von HTML-Elementen für die Übersetzung markieren. Dazu kombinieren Sie `i18n` und den gewünschten Attributnamen:

```

```

## Strings in der Komponentenklasse markieren

Ursprünglich beschränkte sich die Compiler-I18N auf Texte in den Templates. Das war leider für einige Anwendungsfälle zu kurz gedacht. Komponenten könnten beispielsweise Informationen oder Fehlermeldungen in der Komponentenklasse kreieren und diese dann per Datenbindung anzeigen. In diesem Fall benötigen wir jedoch die Möglichkeit, Strings der Komponentenklasse an die jeweiligen Sprachen anzupassen.

Seit Angular 9 ist das zum Glück möglich – wenn auch mit einer etwas gewöhnungsbedürftigen Schreibweise (siehe Beispiel 16-2).

*Beispiel 16-2: Mit `$localize` lassen sich Strings in Komponenten für die Übersetzung markieren.*

```
// src/app/flight-search/flight-search.component.ts

[...]

@Component({ ... })
export class FlightSearchComponent implements OnInit {

[...]

// Hinzufügen:
// ($localize muss NICHT importiert werden!)
info = $localize `:meaning|description@flightSearch-info>Hello World!`;

[...]

ngOnInit(): void {
  console.debug('info', this.info);
}

[...]

}
```

Um Strings an verschiedene Sprachen anzupassen, sind sie als sogenannte *Tagged Strings* einzurichten. Dabei handelt es sich um Strings, die zum einen in Backticks stehen und zum anderen eine Funktion, die den String anpasst, vorangestellt bekommt. Bei dieser Funktion handelt es sich hier um `$localize`.

Bitte beachten Sie, dass Sie `$localize` nicht importieren müssen. Das Paket `@angular/localize` richtet sie als globale Funktion ein.

Der Inhalt des Strings beginnt mit den oben besprochenen Metadaten, die zwischen zwei Doppelpunkten zu halten sind. Danach kommt der Standardwert des Strings.

Diese auf den ersten Blick etwas spezielle Vorgehensweise ist notwendig, da die Compiler-I18N die Texte aus Performancegründen direkt in den Bundles tauscht. Deswegen können wir auch nicht bei Bedarf die Texte zur Laufzeit laden.

## Texte extrahieren

Nachdem Sie alle zu übersetzen Texte markiert haben, können Sie sie mit dem Compiler extrahieren. Dazu rufen Sie die folgende CLI-Anweisung auf:

```
ng extract-i18n
```



In früheren CLI-Versionen kam stattdessen diese mittlerweile abgekündigte Alternative zum Einsatz:

```
ng xi18n
```

Standardmäßig richtet die CLI eine Datei *messages.xlf* ein, die im XLIFF-Format vorliegt. Mit dem Schalter `--format` können Sie jedoch eine der anderen unterstützten Formate anfordern:

```
ng extract-i18n --format xmb  
ng extract-i18n --format arb
```

Die generierten Dateien legt die CLI im Projekthauptverzeichnis ab. Um die Übersicht zu wahren, erzeugen wir dort einen Unterordner *compiler-i18n*, in den wir die Datei *messages.xlf* verschieben. Außerdem duplizieren wir in diesem Ordner die Datei *messages.xlf* und nennen das Duplikat *messages.de.xlf* (siehe Abbildung 16-2).

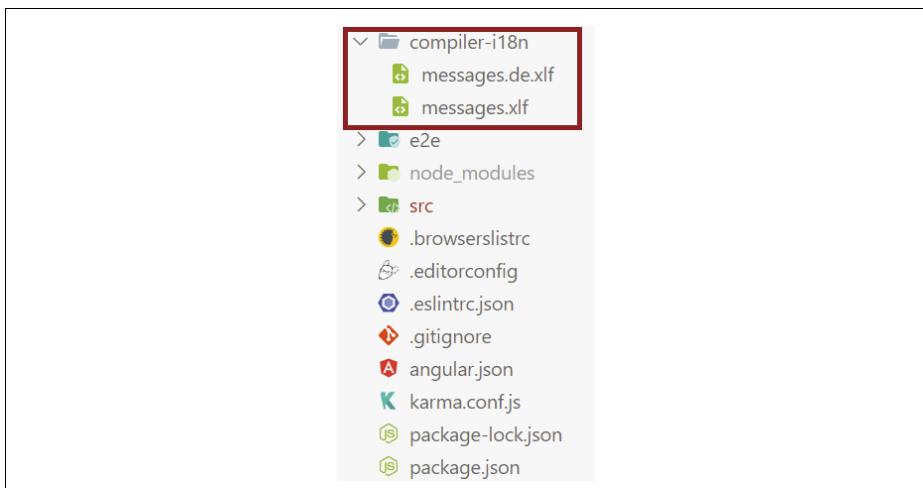


Abbildung 16-2: Ordner mit Übersetzungsdateien

Ein Blick in die Datei *messages.de.xlf* zeigt, dass sie pro extrahiertem Text eine Translation Unit (Element *trans-unit*) aufweist (siehe Beispiel 16-3).

*Beispiel 16-3: Datei mit deutschen Übersetzungen*

```
<?xml version="1.0" encoding="UTF-8" ?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
  <file source-language="en-US" datatype="plaintext" original="ng2.template">
    <body>
      <trans-unit id="flightSearch-info" datatype="html">
        <source>Hello World!</source>

        <!-- Hinzufügen: -->
        <target>Hallo Welt!</target>

        <context-group purpose="location">
          <context context-type="sourcefile">src/app/flight-booking/flight-search/
flight-search.component.ts</context>
          <context context-type="linenumber">18</context>
        </context-group>
        <note priority="1" from="description">description</note>
        <note priority="1" from="meaning">meaning</note>
      </trans-unit>
      <trans-unit id="flightSearch-title" datatype="html">
        <source>Flight Search</source>

        <!-- Hinzufügen: -->
        <target>Flüge suchen</target>

        <context-group purpose="location">
          <context context-type="sourcefile">src/app/flight-booking/flight-search/
flight-search.component.html</context>
          <context context-type="linenumber">3</context>
        </context-group>
        <note priority="1" from="description">Description</note>
        <note priority="1" from="meaning">Meaning</note>
      </trans-unit>
      <trans-unit id="flightSearch-flightsFound" datatype="html">
        <source><x id="INTERPOLATION" equiv-text="      &lt;/button&gt;
"/> flights found!</source>

        <!-- Hinzufügen: -->
        <target><x id="INTERPOLATION" equiv-text="      &lt;/button&gt;
"/> Flüge gefunden!</target>

        <context-group purpose="location">
          <context context-type="sourcefile">src/app/flight-booking/flight-search/
flight-search.component.html</context>
          <context context-type="linenumber">67</context>
        </context-group>
      </trans-unit>
    </body>
  </file>
</xliff>
```

```

        </trans-unit>
    </body>
</file>
</xliff>
```

Neben den Metadaten wie ID, Beschreibung und Bedeutung findet sich darin der ursprüngliche Text in einem Knoten `source`. Wir müssen nun in jeder Translation Unit lediglich einen weiteren Knoten `target` mit der gewünschten Übersetzung platzieren.

Zur Vereinfachung machen wir das hier von Hand. In der Praxis nutzen Übersetzungsstudios dafür in der Regel eigene Softwarepakete.

## Übersetzte Texte in Builds integrieren

Nun sind die Übersetzungsdateien in der `angular.json` zu registrieren. Dazu richten Sie im Knoten `projects/IhrProjektName` einen Kindknoten `i18n` ein (siehe Beispiel 16-4).

*Beispiel 16-4: angular.json mit Übersetzungsdateien*

```
{
  ...
  "projects": {
    "flight-app": {
      "projectType": "application",
      ...
      "root": "",
      "sourceRoot": "src",
      "prefix": "app",
      "i18n": {
        "locales": {
          "de": "compiler-i18n/messages.de.xlf"
        },
        "sourceLocale": "en-US"
      },
      ...
    }
  }
}
```

Die Eigenschaft `locales` bildet die Namen der zu unterstützenden Locales auf ihre Übersetzungsdateien ab. Das `sourceLocale` gibt hingegen an, in welcher Sprache die Templates vorliegen, wenn sie nicht übersetzt werden. Zur Veranschaulichung nutzen wir hier `en-US`.

Nun können Sie die einzelnen Sprachversionen von Compiler und CLI bauen lassen:

```
ng build --localize
```

Sie erhalten jetzt im `dist`-Ordner Ihrer Anwendung pro Locale einen Unterordner (siehe Abbildung 16-3).

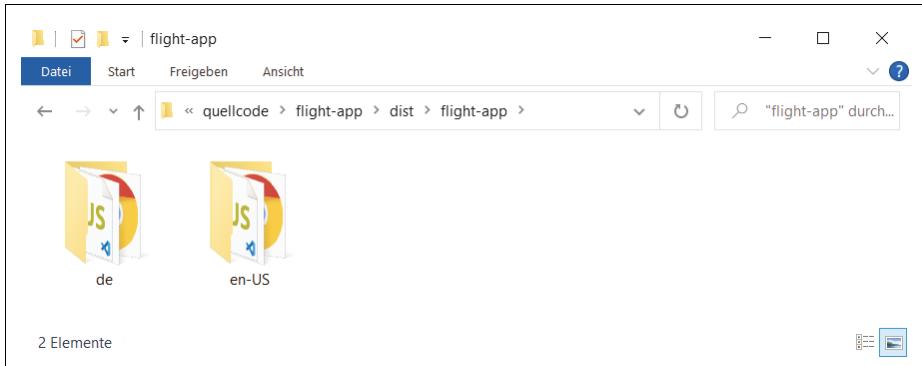


Abbildung 16-3: Ein Ordner pro Locale unter dist

In jedem Ordner befindet sich die gesamte Anwendung mit den jeweiligen Übersetzungen.

Sie können diese Versionen mit einem einfachen kommandozeilenbasierten Webserver ausprobieren. Wir nutzen beispielsweise das npm-Paket serve dazu. Sie können es mit der folgenden Anweisung beziehen:

```
npm i -g serve
```

Wechseln Sie zum Ausprobieren in den *dist*-Ordner Ihrer Anwendung. Das ist jener Ordner, der nun die beiden Unterordner *de* und *en-US* enthält. Dort können Sie serve aufrufen (siehe Abbildung 16-4).

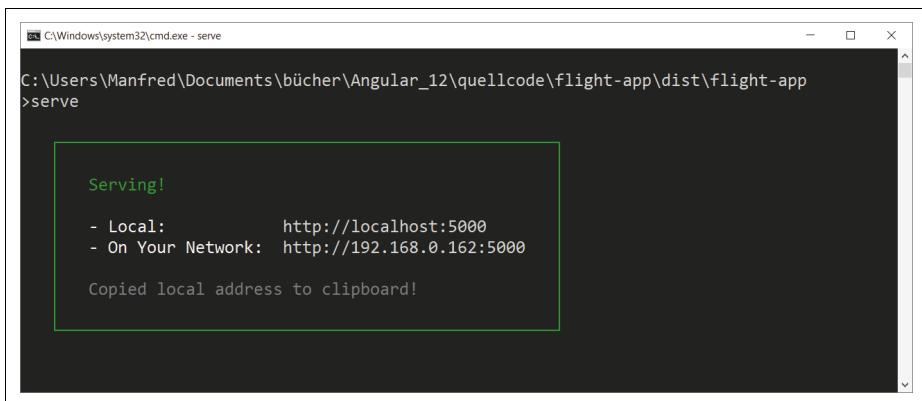


Abbildung 16-4: Kommandozeilenwebserver in dist-Ordner der Anwendung starten

Wenn Sie im Browser auf die von serve verwendete Adresse wechseln (in unserem Fall war es *http://localhost:5000*), sehen Sie das Verzeichnis-Listing mit den beiden Foldern für *de* und *en-US* (siehe Abbildung 16-5).



Abbildung 16-5: Sprachauswahl im Browser

Ein Klick auf *de* führt zur deutschen Sprachversion (siehe Abbildung 16-6).

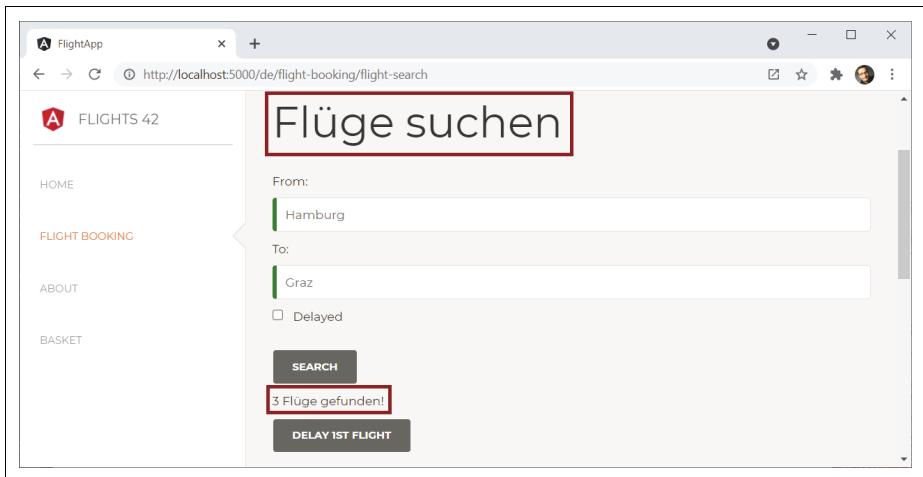


Abbildung 16-6: Deutsche Sprachversion

## Sprache beim Einsatz von ng serve festlegen

Wahrscheinlich wollen Sie auch einzelne Sprachversionen mit `ng serve` ausprobieren. Leider weist `ng serve` keinen Schalter zum Festlegen der gewünschten Sprache auf.

Sie können diese Einschränkung jedoch mit einer Kombination aus einer CLI-Konfiguration für `ng build` und einer für `ng serve` in Ihrer `angular.json` umgehen. Erstere legt die gewünschte Sprache fest:

```
{  
  [...]  
  "projects": {  
    "flight-app": {  
      [...]  
    }  
  }  
}
```

```
"architect": {  
  "build": {  
    [...]  
    "configurations": {  
      "de": {  
        "localize": ["de"]  
      },  
      [...]  
    }  
  }  
}
```

Die Konfiguration für `ng serve` kann zwar selbst keine Sprache festlegen, aber auf die zuvor eingerichtete Konfiguration für `ng build` verweisen:

```
{  
  [...]  
  "projects": {  
    "flight-app": {  
      [...]  
      "architect": {  
        [...]  
        "serve": {  
          [...]  
          "configurations": {  
            "de": {  
              "browserTarget": "flight-app:build:de"  
            }  
            [...]  
          }  
        }  
      }  
    }  
  }  
}
```

Stellen Sie dabei sicher, dass Sie unter `browserTarget` den Namen Ihrer Anwendung, das Wort `build` sowie den Namen der zuvor eingerichteten Konfiguration angeben.

Um nun `ng serve` mit dieser Konfiguration aufzurufen, nutzen Sie folgende Anweisung:

```
ng serve --configuration de
```

Für mehr Sprachen müssen Sie analog dazu weitere Konfigurationen einrichten.

Wir geben zu, auch wir finden diese Vorgehensweise ein wenig kompliziert.

## Übersetzungstexte zur Laufzeit angeben

Obwohl die Compiler-I18N eigentlich die Übersetzungstexte beim Kompilieren in die Bundles integriert, können Sie sie mittlerweile auch im Code setzen. Das muss jedoch vor der ersten Verwendung der einzelnen Texte erfolgen. Da Angular für diese Texte aus Performancegründen keine Datenbindungen einrichtet, ist ein nachträgliches Ändern nach der ersten Verwendung nicht möglich. Hierzu müssen Sie die Anwendung neu in den Browser laden (z.B. mit *F5*).

Um Texte auf diese Weise festzulegen, nutzen Sie die Funktion `loadTranslations`:

```
// src/main.ts  
[...]  
import { loadTranslations } from '@angular/localize';
```

```

loadTranslations({
  'flightSearch-info': 'Liebesgrüße aus der main.ts!',
  'flightSearch-title': 'Flugsuche!!',
  'flightSearch-flightsFound': '{$INTERPOLATION} wurden gefunden!'
});

[...]

```

Der Platzhalter {\$INTERPOLATION} repräsentiert hier die Anzahl der Flüge, die das Template über eine Datenbindung festlegt (siehe »Texte markieren« auf Seite 373). Um bei solchen Strings die Namen der Platzhalter in Erfahrung zu bringen, empfiehlt es sich, zwischenzeitlich die mit i18n markierten Texte in Form von JSON zu extrahieren:

```
ng extract-i18n --format json
```

Die hierdurch generierte *messages.json* beinhaltet die benötigten Schreibweisen.

Sie können die Funktion loadTranslations übrigens auch mehrfach aufrufen, um weitere Texte zu ergänzen. Auf diese Weise kann sogar ein lazy Modul seine eigenen Texte festlegen. Platzieren Sie dazu z.B. den Aufruf von loadTranslations im Konstruktor des Moduls (siehe Beispiel 16-5).

#### *Beispiel 16-5: Übersetzungstexte zur Laufzeit festlegen*

```

// src/app/flight-booking/flight-booking.module.ts

[...]

import { loadTranslations } from '@angular/localize';

@NgModule({
  [...]
})
export class FlightBookingModule {

  constructor() {
    loadTranslations({
      'flightSearch-info': 'Liebesgrüße aus der main.ts!!',
      'flightSearch-title': 'Flugsuche!!',
      'flightSearch-flightsFound': '{$INTERPOLATION} wurden gefunden!!'
    });
  }
}

```



Die an loadTranslations übergebenen Texte können Sie sogar bei Bedarf via HTTP, z.B. mit dem HttpClient, laden. Allerdings müssen Sie sicherstellen, dass die Texte vor ihrer ersten Verwendung vorhanden sind. Da HTTP-Zugriffe asynchron erfolgen, lässt sich das jedoch nicht ohne Weiteres garantieren. Abhilfe schaffen Resolver, die in der Lage sind, einen Routing-Vorgang zu verzögern (siehe Kapitel 8).

## Grammatikalische Formen berücksichtigen

Das Austauschen von Texten ist nur eine Aufgabe, die bei der Internationalisierung einer Anwendung anfällt. Ein weiterer Aspekt ist das Berücksichtigen von unterschiedlichen grammatischen Formen, etwa bei unterschiedlichen Geschletern oder bei der Unterscheidung von Einzahl und Mehrzahl. Dabei gilt es auch zu beachten, dass verschiedene Sprachen unterschiedlich viele Geschlechter und Mehrzahlformen kennen.

Als Beispiel für die Unterscheidung zwischen Einzahl und Mehrzahl greifen wir hier erneut die `FlightSearchComponent` auf. Wenn diese über die Anzahl an gefundenen Flügen informiert, sollte von »1 Flug«, aber von »3 Flügen« die Rede sein. Die `Compiler-I18N` sieht hierfür eine eigene Grammatik im Template vor:

```
<!-- src/app/flight-search/flight-search.component.html -->
[...]
<div i18n="@@flightSearch-flightsFound">
  {flights.length, plural, =1 {1 flight found} other
   {{flights.length}} flights found}
</div>
[...]
```

Hier legen wir für den Wert `1` einen Text sowie für alle anderen Werte (`other`) einen weiteren Text fest. Die CLI und der Angular-Compiler extrahieren den gesamten Ausdruck in die Sprachdateien. Somit können Sie bei einer Übersetzung in einer Sprache mit mehreren Pluralformen zum Beispiel auch einen eigenen Text für die Werte `2` bis `5` angeben.



Neben der Angabe von Texten für einen konkreten Wert (z.B. `=1`) und alle anderen (`other`) unterstützt Angular auch die folgenden Kategorien: `zero`, `one`, `two`, `few`, `many` und `other`.

Um die Unterscheidung verschiedener Geschlechtsformen zu demonstrieren, erweitern wir die `PassengerSearchComponent` um eine Eigenschaft `passenger` (siehe Beispiel 16-6).

*Beispiel 16-6: PassengerSearchComponent mit Passagier*

```
// src/app/flight-booking/passenger-search/passenger-search.component.ts

[...]

@Component(...)
export class PassengerSearchComponent implements OnInit {

  [...]

  // Hinzufügen:
  passenger = {
    name: 'Max',
    gender: 'male'
  };
}
```

```
[...]  
ngOnInit(): void {  
}  
}
```

Das dazugehörige Template könnte nun wie das in Beispiel 16-7 aussehen.

*Beispiel 16-7: Die PassengerSearchComponent unterscheidet verschiedene Geschlechter.*

```
<!-- src/app/flight-booking/passenger-search/passenger-search.component.html -->
```

```
[...]
```

```
<!-- Hinzufügen -->  
<p i18n="@@passenger-bookedTicket">  
    You've booked a ticket for {{passenger.name}}  
</p>  
<p i18n="@@passenger-forward">  
{passenger.gender, select,  
    male {Forward it to him.}  
    female {Forward it to her.}  
    other {Forward it to them.}}  
</p>
```



Falls Sie solche Texte, wie unter »Übersetzungstexte zur Laufzeit angeben« gezeigt, zur Laufzeit festlegen, stehen Sie mitunter vor der Herausforderung, nicht zu wissen, wie diese Fallunterscheidungen für loadTranslations darzustellen sind. Abhilfe schafft ein Export der Texte als JSON:

```
ng extract-i18n --format json
```

Die hierdurch erzeugte *messages.json* enthält die Texte in der von loadTranslations erwarteten Form.

## Unterschiedliche Formate unterstützen

Neben dem Austauschen von Sprachtexten und dem Berücksichtigen unterschiedlicher grammatischer Formen gilt es auch, die lokalen Gepflogenheiten zur Formatierung von Zahlen und Datumswerten zu beachten.

Um das zu ermöglichen, kommt Angular mit Metadaten zu allen im *Unicode Common Locale Data Repository* (<http://cldr.unicode.org>) hinterlegten Locales. Beim Build ergänzt die CLI die gebauten Sprachversionen um diese Metadaten.

Pipes wie date oder number greifen diese Metadaten zur Laufzeit auf. Damit das funktioniert, müssen Sie jedoch bei der date-Pipe auf logische Datumsformate setzen. Beispiele dafür sind short, medium oder large für verschieden ausführliche Schreibweisen:

```
<!-- src/app/flight-booking/flight-card/flight-card.component.html -->
```

```
[...]
```

```
<p>Date: {{item?.date | date:'short'}}</p>
[...]
```

Bei Pipes für Zahlen ist keine Anpassung vorzunehmen. Sie nutzen ohne weiteres Zutun das Tausender- und das Dezimaltrennzeichen des aktuellen Locales.

Mehr ist nicht notwendig, damit die mit `ng build --localize` gebauten Sprachversionen das richtige Format berücksichtigen (siehe Abbildung 16-7).

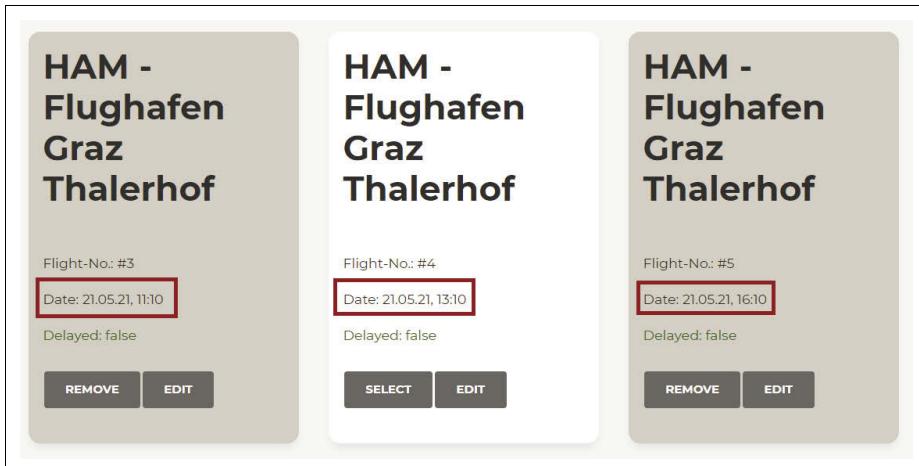


Abbildung 16-7: Deutsche Sprachversion mit deutschem Datumsformat



Pipes eignen sich leider nur für die Ausgabe. Wollen Sie auch die Eingabe lokalisieren, müssen Sie auf entsprechende Steuerelemente von Dritten zurückgreifen. Das freie Angular Material (<https://material.angular.io/>) bietet zum Beispiel einen Dateipicker.

Alternativ dazu können Sie auch Ihren eigenen ControlValueAccessor, der das gewünschte EingabefORMAT unterstützt, schreiben (siehe Kapitel 18).

## Manuell weitere Formate laden

Falls Sie weitere Formate unterstützen möchten oder selbst ohne Einsatz der Compiler-API auf die in Angular inkludierten Metadaten für diese Formate zurückgreifen wollen, können Sie diese beim Anwendungsstart auch selbst laden.

Dazu importieren Sie die Metadaten aus dem Namensraum `@angular/common/locales/xy`, wobei `xy` für das gewünschte Locale steht:

```
// src/main.ts

// Hinzufügen:
import ES from '@angular/common/locales/es';
import deAT from '@angular/common/locales/de-AT';
import { registerLocaleData } from '@angular/common';
```

```
// Vor Bootstrapping hinzufügen:  
registerLocaleData(DE);  
registerLocaleData(deAT);
```

Diese Metadaten übergeben Sie anschließend an registerLocaleData.

Um nun eines der geladenen Locales zu nutzen, übergeben Sie es an den letzten Parameter der jeweiligen Pipe:

```
<p>Date: {{item?.date | date:'long':'es'}}</p>
```

Bitte beachten Sie bei der date-Pipe, dass sie drei Parameter aufweist. Diese spiegeln das Format, ein eventuell manuell festzulegendes Zeitzonenoffset sowie das zu nutzende Locale. Nutzen Sie das Zeitzonenoffset nicht, können Sie es auf einen Leerstring setzen.

## ngx-translate

Die zuvor betrachtete Compiler-I18N zeichnet sich vor allem durch ihre gute Performance aus. Da sie die Übersetzungstexte in die Bundles einwebt, muss die Angular-Anwendung zur Laufzeit keinen Aufwand für das Laden oder für die Datenbindung investieren.

Diese gute Performance erkauft sich Angular jedoch durch eine verringerte Flexibilität: Aufgrund des bewussten Verzichts auf die Nutzung der Datenbindung erlaubt die Compiler-I18N keine Sprachwechsel zur Laufzeit. Stattdessen muss sie mit einem Hyperlink auf die gewünschte Sprachversion verweisen.

Das populäre Community-Projekt ngx-translate dreht die Vor- und Nachteile dieses Ansatzes um. Es erlaubt das Nachladen und Wechseln der Sprache zur Laufzeit, setzt dafür aber auf Datenbindung. Letzteres kann sich (theoretisch) auf die Laufzeitperformance auswirken.

## Überblick

Die Bibliothek ngx-translate lädt bei Bedarf Sprachdateien und platziert deren Texte per Datenbindung in den Templates (siehe Abbildung 16-8).

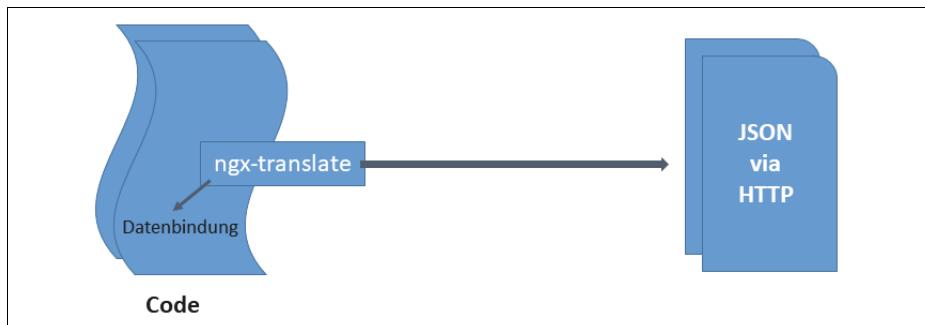


Abbildung 16-8: Funktionsweise von ngx-translate

Standardmäßig bezieht ngx-translate die Texte in Form von JSON via HTTP. Aufgrund des modularen Aufbaus der Bibliothek lassen sich jedoch diese beiden Aspekte austauschen, sodass Sie auch andere Datenformate und Transportwege nutzen können.

Der Einsatz der Datenbindung wirkt sich theoretisch auf die Laufzeitperformance der Anwendung aus. In unserer Praxis können wir dieses Problem jedoch nicht beobachten, obwohl ein großer Teil unserer Kunden auf ngx-translate setzt. Die Datenbindung ist bei Angular an sich sehr gut optimiert, und Maßnahmen wie OnPush erhöhen zusätzlich ihre Effizienz (siehe Kapitel 13).

## Bibliothek installieren und konfigurieren

Die Bibliothek ngx-translate lässt sich via npm installieren:

```
npm install @ngx-translate/core --save
```

Außerdem müssen Sie einen Loader für die Bibliothek einrichten. Dabei handelt es sich um einen austauschbaren Service, der sich um das Laden der Texte kümmert. Die Standardimplementierung lässt sich ebenfalls über npm beziehen:

```
npm install @ngx-translate/http-loader --save
```

Danach importieren Sie das von der Bibliothek bereitgestellte TranslateModule in Ihr AppModule (siehe Beispiel 16-8).

*Beispiel 16-8: TranslateModule importieren*

```
// src/app/app.module.ts

[...]

// Hinzufügen:
import { HttpClient } from '@angular/common/http';
import { TranslateHttpLoader } from '@ngx-translate/http-loader';
import { TranslateLoader, TranslateModule } from '@ngx-translate/core';

export function createLoader(http: HttpClient) {
    return new TranslateHttpLoader(http, '/assets/i18n/', '.json');
}

@NgModule({
    imports: [
        ...
        HttpClientModule,
        // Hinzufügen:
        TranslateModule.forRoot({
            defaultLanguage: 'en',
            useDefaultLang: true,
            loader: {
                provide: TranslateLoader,

```

```

        useFactory: createLoader,
        deps: [HttpClient]
    },
    ...
],
...
})
export class AppModule { }

```

Die `forRoot`-Methode von `TranslateModule` erlaubt das Konfigurieren des Moduls. Wir nutzen hier die folgenden Optionen:

#### *defaultLanguage*

Die Standardsprache, die als Fallback zu nutzen ist. Liegt eine Übersetzung in der gerade aktivierte Sprache nicht vor, bezieht `ngx-translate` den gewünschten Text aus der Sprachdatei der Standardsprache.

#### *useDefaultLang*

Legt fest, ob die Standardsprache initial aktiviert werden soll. Alternativ dazu könnten Sie, wie weiter unten gezeigt, mit dem `TranslateService` eine andere Sprache im Zuge des Programmstarts aktivieren.

#### *loader*

Hierbei handelt es sich um einen Provider, der den Loader bereitstellt.

Den `TranslateHttpLoader` erzeugt die Anwendung über eine Factory. Sie nimmt den `HttpClient` entgegen und reicht ihn an den Loader weiter. Außerdem legt sie ein Präfix und ein Suffix für die Sprachdateien fest, nämlich `/assets/i18n/` sowie `.json`. Das hat zur Folge, dass der Loader englische Texte in der Datei `/assets/i18n/en.json` erwartet. Für ihre deutschen Gegenstücke lädt der Loader hingegen `/assets/i18n/de.json`.

Neben dem `AppModule` muss auch jedes andere Modul, das `ngx-translate` nutzen möchte, das `TranslateModule` importieren. In unserem Beispiel ist das beim `FlightBookingModule` der Fall (siehe Beispiel 16-9).

*Beispiel 16-9: Auch das FlightBookingModule erhält das TranslateModule.*

```

// src/app/flight-booking/flight-booking.module.ts

[...]
// Hinzufügen:
import { TranslateModule } from '@ngx-translate/core';

@NgModule({
  imports: [
    ...
    // Hinzufügen:
    TranslateModule.forChild(),
  ],
  ...
})
export class FlightBookingModule { }

```

Gemäß den üblichen Konventionen in der Angular-Welt kommt hierfür die Methode `forChild` zum Einsatz.

## Sprachdateien bereitstellen

Für die Sprachdateien erzeugen wir unter `assets` einen Ordner `i18n`. Wir nennen sie `en.json` (siehe Beispiel 16-10) und `de.json` (siehe Beispiel 16-11).

*Beispiel 16-10: Englische Texte in assets/i18n/en.json*

```
{  
  "flights": {  
    "title": "Flight Search",  
    "found": "{{count}} flights found",  
    "info": "Hello World!"  
  }  
}
```

*Beispiel 16-11: Deutsche Texte in assets/i18n/de.json*

```
{  
  "flights": {  
    "title": "Flugsuche",  
    "found": "{{count}} Flüge gefunden",  
    "info": "Hallo Welt!"  
  }  
}
```

Die Sprachdateien können beliebig viele Zwischenebenen, wie z.B. `flights`, aufweisen. Außerdem können die Texte auch Platzhalter nutzen. Diesen wird zur Laufzeit ein konkreter Wert zugewiesen. Ein Beispiel dafür ist `count` im Eintrag `found`.

## Texte einbinden

Um einen direkten Vergleich zur Compiler-I18N zu ermöglichen, nutzen wir hier `ngx-translate` an den gleichen Stellen. Die `FlightSearchComponent` ist dazu wie in Beispiel 16-12 anzupassen.

*Beispiel 16-12: Übersetzungen abrufen*

```
<!-- src/app/flight-search/flight-search.component.html -->  
  
<h1>{{ 'flights.title' | translate }}</h1>  
  
[...]  
  
<div class="form-group" *ngIf="flights$ | async as flights">  
  
  [...]  
  
  <div>  
    {{ 'flights.found' | translate:{ count: flights.length } }}  
  </div>
```

```
[...]
```

```
</div>
```

Zum Beziehen der Texte kommt die translate-Pipe zum Einsatz. Ihr wird der Schlüssel des gewünschten Werts, z.B. flights.title, übergeben. Als optionalen Parameter erlaubt sie ein Objekt mit Werten für Platzhalter.



Die translate-Pipe funktioniert intern ähnlich wie die von Angular gelieferte async-Pipe. Sie informiert Angular über neue Werte und beschleunigt somit die Datenbindung im OnPush-Modus (siehe Kapitel 13).

## Texte zur Laufzeit laden

Mit ngx-translate können Sie auch direkt im Programmcode die gewünschten Übersetzungstexte beziehen. Dazu bietet der TranslateService die Methode get an (siehe Beispiel 16-13).

*Beispiel 16-13: Übersetzungstexte direkt im Programmcode beziehen*

```
// src/app/flight-search/flight-search.component.ts

[...]

// Hinzufügen:
import { TranslateService } from '@ngx-translate/core';
import { OnDestroy } from '@angular/core';

@Component(...)
// OnDestroy hinzufügen:
export class FlightSearchComponent implements OnInit, OnDestroy {

    // Hinzufügen:
    closeSubject = new Subject<void>();

[...]

constructor(
    // TranslateService injizieren lassen:
    private translate: TranslateService,
    private flightService: FlightService) {
}

// Aktualisieren:
ngOnInit(): void {
    this.translate.get('flights.info').pipe(
        takeUntil(this.closeSubject)
    )
    .subscribe(
        info => console.debug('info', info)
    );
}

this.translate.get('flights.found', { count: 0 }).pipe(
```

```

        takeUntil(this.closeSubject)
    )
.subscribe(
    found => console.debug('found', found)
);
}

// Hinzufügen:
ngOnDestroy(): void {
    this.closeSubject.next();
}

[...]
}

```

Die Methode `get` nimmt den Schlüssel des gewünschten Texts sowie optional ein Objekt mit Werten für Platzhalter entgegen. Das zurückgelieferte Observable liefert danach augenblicklich den Text in der aktuellen Sprache. Bei jedem Sprachwechsel liefert es den aktualisierten Text.

Das Zusammenspiel zwischen `closeSubject`, `ngOnDestroy` und `takeUntil` stellt sicher, dass die Anwendung das Observable beim Verlassen der Komponente schließt (siehe Kapitel 11).

## Sprachwechsel

Für den Sprachwechsel bietet der `TranslateService` die Methode `use` an (siehe Beispiel 16-14).

*Beispiel 16-14: Sprache zur Laufzeit ändern*

```

// src/app/navbar/navbar.component.ts

[...]

// Hinzufügen:
import { TranslateService } from '@ngx-translate/core';

@Component(...)
export class NavbarComponent {

    [...]

    // Aktualisieren:
    constructor(private translate: TranslateService) {
    }

    // Hinzufügen:
    setLang(lang: string): void {
        this.translate.use(lang);
    }

    [...]
}

```

In unserem Beispiel ruft das Template der NavBar die hier gezeigte setLang-Methode auf und übergibt die gewünschte Sprache:

```
<!-- src/app/navbar/navbar.component.html -->

[...]

<ul class="navbar-nav">

  <li class="nav-item">
    <a (click)="setLang('en')" class="nav-link">EN</a>
  </li>

  <li class="nav-item">
    <a (click)="setLang('de')" class="nav-link">DE</a>
  </li>

  [...]
</ul>

[...]
```

Diese Erweiterung gibt uns zwei Hyperlinks, die oben rechts erscheinen und uns die Möglichkeit bieten, zwischen Deutsch und Englisch zu wechseln (siehe Abbildung 16-9).

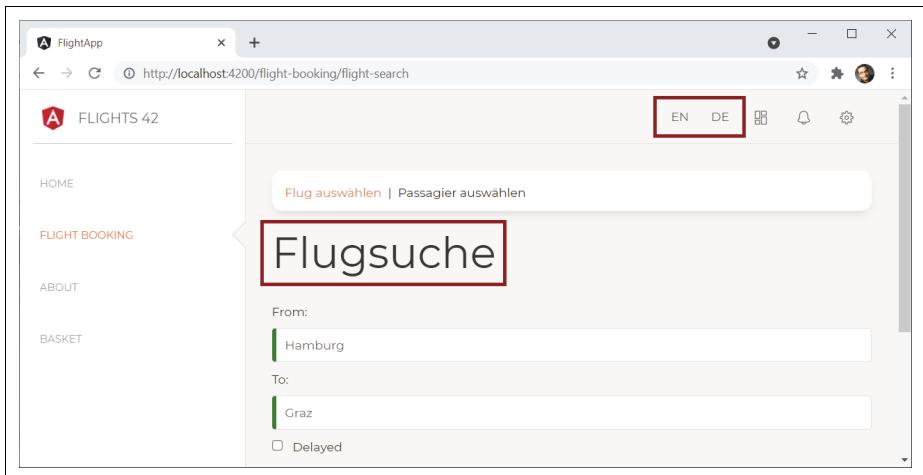


Abbildung 16-9: Links für einen Sprachwechsel zur Laufzeit

## Grammatikalische Formen berücksichtigen

Unterschiedliche grammatischen Formen kann die Bibliothek ngx-translate nicht direkt berücksichtigen. Allerdings lässt sie sich mit zwei Pipes, die im Lieferumfang von Angular inkludiert sind, kombinieren. Die Pipe `i18nPlural` erlaubt den Wechsel zwischen Einzahl und Mehrzahl, und mit `i18nSelect` können Sie unterschiedliche Geschlechter berücksichtigen.

Die Konfigurationen für diese Pipes lassen sich in den Sprachdateien als Objekte hinterlegen (siehe Beispiel 16-15).

*Beispiel 16-15: Übersetzungsdatei en.json mit Konfigurationen für i18nPlural und i18nSelect*

```
{  
  "flights": {  
    "title": "Flight Search",  
    "info": "Hello World!",  
    "booked": "You've booked a ticket for {{name}}",  
    "found": {  
      "=1": "One flight found",  
      "other": "# flights found"  
    },  
    "forward": {  
      "male": "Forward it to him",  
      "female": "Forward it to her",  
      "other": "Forward it to them"  
    }  
  }  
}
```

Der Knoten `found` enthält eine Variante für einen einzigen gefundenen Flug (=1) und eine weitere für alle anderen Fälle (other). Die Raute (#) ist ein Platzhalter für den verwendeten numerischen Wert. Unter `forward` finden wir die gleiche Information für verschiedene Geschlechter.

Zum Ausprobieren der `i18nPlural`-Pipe bietet sich das Template der `FlightSearch` Component an:

```
<!-- src/app/flight-search/flight-search.component.html -->  
[...]  


{{ flights.length | i18nPlural: ('flights.found' | translate)}  
</div>  
[...]


```

Der übergebene Wert ist die zu nutzende Zahl. Als Parameter hängen wir zusätzlich die Konfiguration aus der Sprachdatei an. Diese beziehen wir mit der `translate`-Pipe. Das zeigt, dass die `translate`-Pipe in der Lage ist, ganze Objekte aus der Sprachdatei auszulesen.

Das Anwenden von `i18nSelect` funktioniert sehr ähnlich. Zur Demonstration nutzen wir hier wieder die `PassengerSearchComponent` mit der oben eingeführten Eigenschaft `passenger` (siehe Beispiel 16-16).

*Beispiel 16-16: PassengerSearchComponent mit der Eigenschaft passenger*

```
[...]
```

```
@Component({ ... })
```

```

export class PassengerSearchComponent implements OnInit {
    [...]
    passenger = {
        name: 'Susi',
        gender: 'female'
    };
    [...]
}

```

Im dazugehörigen Template setzen wir die `i18nSelect`-Pipe ein:

```

<!-- src/app/flight-booking/passenger-search/passenger-search.component.html -->
[...]
<p>{{ 'flights.booked' | translate: { name: passenger.name } }}</p>
<p>{{ passenger.gender | i18nSelect:('flights.forward' | translate) }} </p>

```

Die Pipe `i18nSelect` nimmt den Wert, anhand dessen die Fallunterscheidung erfolgt, entgegen. Hier ist das `passenger.gender`. Als Parameter hängt das Beispiel die Konfiguration aus der Sprachdatei an. Auch hier lesen wir sie mit der `translate`-Pipe aus.

## Unterschiedliche Formate nutzen

Auch für den Einsatz unterschiedlicher Formate bringt `ngx-translate` keine Unterstützung. Allerdings lässt sich hier erneut ein Bordmittel von Angular nutzen, nämlich die Möglichkeit, Metadaten für Formate programmatisch zu laden (siehe »Manuell weitere Formate laden« auf Seite 384).

## Lazy Loading

Verwenden Sie Lazy Loading, wollen Sie eventuell Übersetzungstexte erst gemeinsam mit dem jeweiligen Modul laden. Um das zu ermöglichen, spendieren Sie dem lazy Modul einen eigenen Loader, der auf die gewünschte Sprachdatei verweist (siehe Beispiel 16-17).

*Beispiel 16-17: Lazy Loading von Übersetzungstexten*

```

// src/app/flight-booking/flight-booking.module.ts
[...]
// Aktualisieren:
import { TranslateLoader, TranslateModule, TranslateService }
    from '@ngx-translate/core';

// Hinzufügen:
import { TranslateHttpLoader } from '@ngx-translate/http-loader';

```

```

// Loader lädt nun Dateien aus dem Unterordner flight-booking.
export function createLoader(http: HttpClient) {
  return new TranslateHttpLoader(http, '/assets/i18n/flight-booking/', '.json');
}

@NgModule({
  imports: [
    [...]
    TranslateModule.forChild({
      loader: {
        provide: TranslateLoader,
        useFactory: createLoader,
        deps: [HttpClient]
      },
      defaultLanguage: 'en',
      isolate: true,
    }),
    [...]
  ],
  ...
})
export class FlightBookingModule {
}

```

In diesem Beispiel wird der Loader angewiesen, die Texte für das FlightBooking Module aus dem Unterordner *flight-booking* zu laden.

Damit der Loader tätig wird, ist die Eigenschaft `isolate` auf `true` zu setzen. Damit isolieren Sie das FlightBookingModule vom Parent-Modul, und das hat zur Folge, dass der Loader seine eigenen Übersetzungstexte lädt.

Bevor Sie Ihre Lösung ausprobieren, sollten Sie noch entsprechende Übersetzungsdateien im Ordner `/assets/i18n/flight-booking` ablegen und ng serve zur Sicherheit neu starten.



Leider haben Sie beim Einsatz von `isolate` keinen Zugriff mehr auf die Übersetzungstexte des Parent-Moduls. Über dieses Problem diskutiert die Community rund um `ngx-translate` schon länger, und es existieren auch ein paar Workarounds. Einen davon finden Sie in unseren Buchbeispielen im Branch `16-I18N-ngx-translate-lazy`. Bei Interesse werfen Sie einen Blick in den Konstruktor des `FlightBookingModule`, der mit einer Hilfsmethode die Übersetzungen aus dem Parent-Modul ins aktuelle kopiert.

## Zusammenfassung

Eines der wichtigsten Architekturziele des Angular-Teams ist Performance. Dieses Ziel wirkt sich auch auf die integrierte I18N-Lösung aus: Sie basiert auf dem Angular-Compiler, der zum einen Texte extrahiert und diese zum anderen mit ihren Übersetzungen tauscht. Somit muss sich Angular zur Laufzeit nicht mit dem Laden und Darstellen der Übersetzungen belasten.

Diese tolle Performance hat jedoch einen Preis: Sie schränkt die Flexibilität dieser sogenannten Compiler-I18N ein. Pro Sprache erzeugt der Angular-Compiler zum Beispiel im Zusammenspiel mit der CLI ein eigenes Bundle-Set. Außerdem kann die Anwendung die Sprache zur Laufzeit nicht ändern. Stattdessen kann sie lediglich Hyperlinks, die auf andere Sprachversionen verweisen, anbieten. Die Vorteile von SPAs gehen an dieser Stelle verloren.

Die beliebte Community-Lösung `ngx-translate` dreht die Vor- und Nachteile der Compiler-I18N um: Sie ist sehr flexibel und erlaubt das Wechseln der Sprache zur Laufzeit. Dafür muss sie jedoch ein wenig Aufwand auf sich nehmen, um die Übersetzungstexte zu laden und sie über Datenbindung zur Anzeige zu bringen.



# Reaktive Zustandsverwaltung mit NGRX (Redux)

Zustandslos und schlank – das sind die Tugenden des wartbaren Weblayers. Wenn gleich diese Daumenregel für klassische serverseitige Webanwendungen stimmen mag, gilt sie nicht mehr für moderne Single Page Applications. Die sind nämlich in den Ring gestiegen, um die Benutzerfreundlichkeit klassischer Desktopanwendungen ins Web zu bringen. Und in puncto Benutzerfreundlichkeit ist es nicht gerade von Vorteil, wenn die Anwendung ständig Daten vom Server nachlädt. Das bedeutet, dass plötzlich eine ganze Menge an Daten lokal vorliegen und der Client doch nicht mehr ganz so schlank ausfällt.

Damit das nicht in einem Chaos mündet, wurden in den letzten Jahren einige Ansätze zur lokalen Zustandsverwaltung entwickelt. Der populärste ist wohl *Redux*, das aus der Welt von React stammt. Mit `@ngrx/store` liegt auch eine Implementierung für Angular vor. Da sie unter anderem von einem Angular Core Team Member entwickelt wurde und auf die in Angular weitverbreiteten Observables setzt, kommt ihr der Stellenwert eines De-facto-Standards zu.

In diesem Kapitel zeigen wir, warum Redux und `@ngrx/store` Sinn ergeben und wie sie funktionieren.

## Zustandsverwaltung mit Services

Zur Verwaltung von Zuständen gibt uns Angular Services an die Hand. Da sie Singletons in ihrem Wertebereich sind, können sie Daten zentral vorhalten und als Drehscheibe für einzelne Komponenten dienen. Solange die Interaktion mit ihnen wie in Abbildung 17-1 aussieht, ist auch alles wunderbar.

Die Erfahrung hat jedoch gezeigt, dass selbst bei kleineren Anwendungen die Daten eines Service an vielen Stellen der Anwendung benötigt werden. Somit ergibt sich eine verworrene Struktur mit schwer nachvollziehbaren Datenflüssen (siehe Abbildung 17-2).

Besonders undurchsichtig wird es, wenn sich Services und Komponenten gegenseitig benachrichtigen, um Daten weiterzugeben. Schnell ergeben sich Zyklen, die die Performance negativ beeinflussen und im schlimmsten Fall die Anwendung einfrie-

ren. Da jeder Service seinen eigenen Ausschnitt des Anwendungszustands hat, kommt es auch zu Redundanzen, die auseinanderlaufen und zu Inkonsistenzen führen.

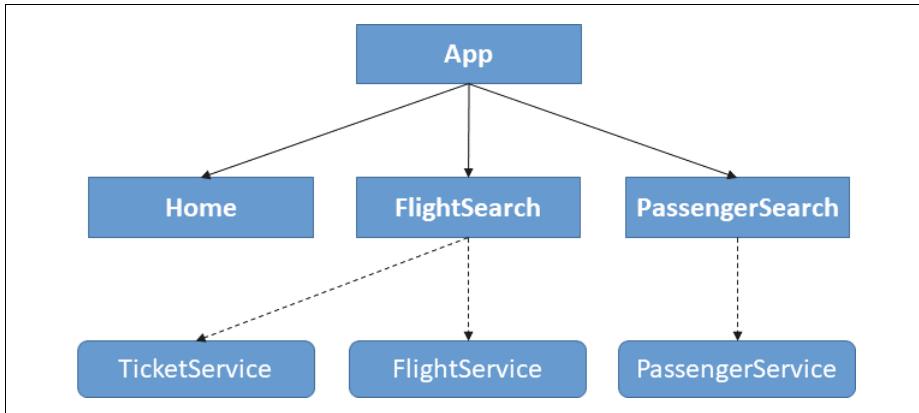


Abbildung 17-1: Komponenten und Services

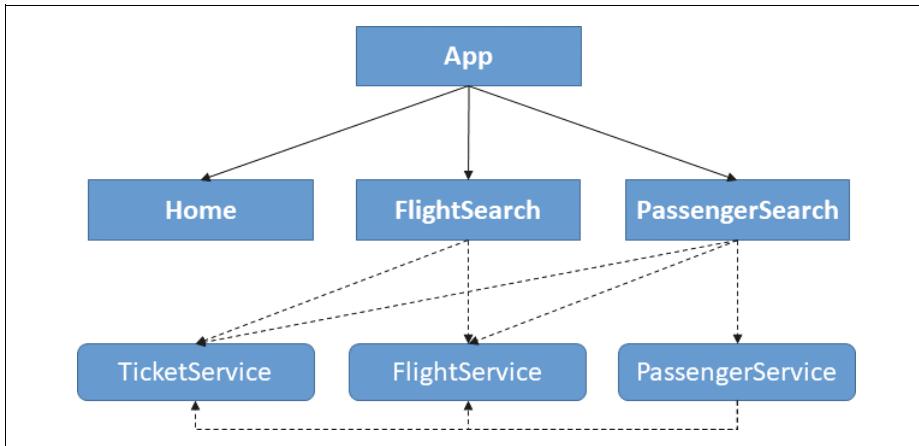


Abbildung 17-2: Verworrne Struktur

Im Fehlerfall kann sich dann niemand mehr erklären, wie es der Benutzer geschafft hat, den jeweiligen Anwendungszustand herbeizuführen. Genau deswegen ist in den letzten Jahren in der React-Community das Redux-Muster entstanden, das nachfolgend anhand der Angular-Implementierung @ngrx/store näher betrachtet wird.

## Das Redux-Muster

Das Muster *Redux* sieht vor, dass der gesamte Anwendungszustand von einem globalen Store verwaltet wird. Diesen könnte man mit einer einfachen In-Memory-Datenbank im Browser vergleichen (siehe Abbildung 17-3).

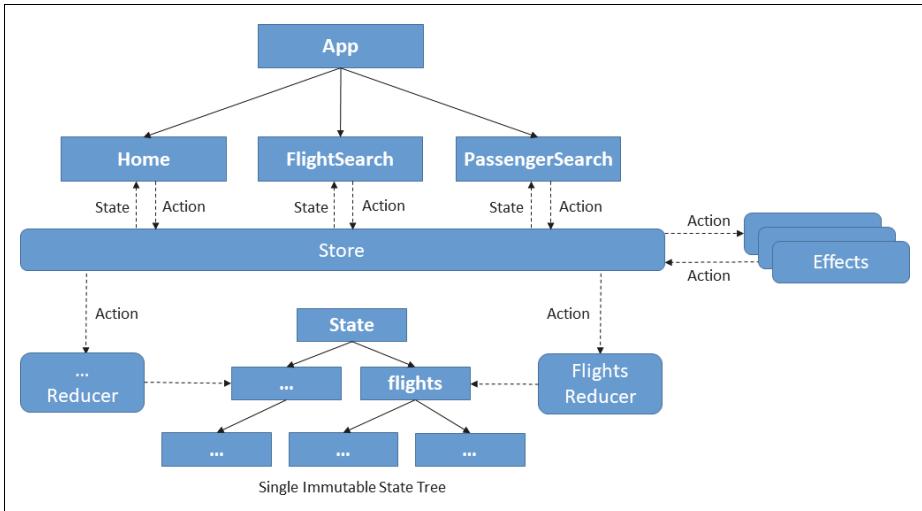


Abbildung 17-3: Redux

Der Store verwaltet den Anwendungszustand in einem Objektbaum. In erster Näherung könnte man sich die Wurzel als Datenbankschema und die Knoten in der Ebene darunter als Tabellen vorstellen. Dieser Zustandsbaum ist unveränderbar (*immutable*), was bedeutet, dass bei jeder Änderung die betroffenen Objekte zu tauschen sind. Somit kann der Store durch Vergleich der Objektreferenzen herausfinden, welche Knoten sich geändert haben, ohne sämtliche Eigenschaften zu überwachen.

Das führt zu einer besseren Performance bei der Änderungsverfolgung. Da beim Lesen nichts kaputtgehen kann, gewährt Redux jedem Systembestandteil einen direkten Lesezugriff. Dazu bietet die hier betrachtete Implementierung `@ngrx/store` Observables an, die den Interessenten bei Datenänderungen informieren.

Direkt schreiben dürfen die einzelnen Komponenten jedoch nicht. Hier wäre die Gefahr von Inkonsistenzen und Redundanzen zu groß. Stattdessen senden sie lediglich Actions an den Store. In erster Näherung könnte man sich eine Action wie den Aufruf einer Stored Procedure vorstellen. Sie beinhaltet einen Typ, den man mit dem Namen der Stored Procedure vergleichen könnte, und eine Payload mit Parametern.

Zum Abarbeiten der Actions finden sich im Store sogenannte Reducer. Dabei handelt es sich um Funktionen, die für einen bestimmten Ast des Zustandsbaums verantwortlich sind. Um Zyklen zu vermeiden, erhält jeder Reducer jede vom Store empfangene Action und kann mit den darin zu findenden Informationen seinen Teil des Zustandsbaums auf den neuesten Stand bringen. Natürlich wird nicht jeder Reducer auf alle Actions reagieren, sondern zunächst mal ermitteln, ob die aktuelle Action für ihn überhaupt relevant ist.

Reducer laufen jedoch immer synchron ab. Für asynchrone Nebeneffekte kommt das Schwesterprojekt `@ngrx/effects` zum Einsatz. Actions können damit außerhalb

des Stores asynchrone Operationen – sogenannte Effects – anstoßen. Sobald diese fertig sind, senden sie eine weitere Action an den Store. Damit signalisieren sie den Erfolg samt Ergebnis oder einen Fehlerzustand.

## NGRX einrichten

Um die Umsetzung des Redux-Musters mit @ngrx/store zu veranschaulichen, fügen wir zunächst die Bibliothek unserer Anwendung hinzu:

```
ng add @ngrx/store
```

Diese Anweisung lädt das Paket @ngrx/store herunter und importiert sein Store Module in das AppModule der Anwendung. Hierzu kommt, wie auch schon vom Router bekannt, seine statische forRoot-Methode zum Einsatz:

```
imports: [
  [...]
  StoreModule.forRoot({}, {}),
]
```

Die beiden Argumente erlauben das Registrieren von Reducern und sogenannten Meta-Reducern. Letzteres sind Reducer, die alle anderen Reducer erweitern. Damit lassen sich zum Beispiel sämtliche Zustandsänderungen protokollieren. Wir lassen diese beiden Argumente fürs erste Mal außen vor, unter anderem weil wir unsere Reducer in ein paar Momenten auf der Ebene des Feature-Module FlightBooking Module einrichten werden.

Das oben erwähnte Schwesterprojekt @ngrx/effects lässt sich auf die gleiche Weise hinzufügen:

```
ng add @ngrx/effects
```

Ähnlich wie zuvor importiert diese Anweisung das EffectsModule ins AppModule der Anwendung:

```
imports: [
  [...]
  EffectsModule.forRoot([]),
],
```

Auch hier ignorieren wir vorerst den übergebenen Parameter. Wir werden uns weiter unten im Rahmen des FlightBookingModule um das Thema Effects kümmern.

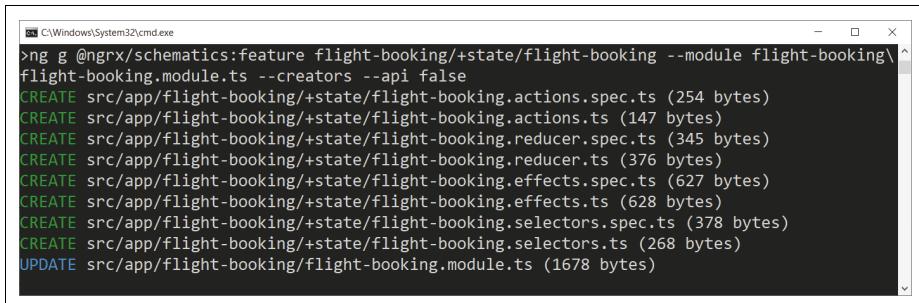
Um uns die Arbeit mit NGRX zu vereinfachen, installieren wir uns die Bibliothek @ngrx/schematics:

```
npm i @ngrx/schematics
```

Diese Bibliothek beinhaltet Baupläne (Schematics), die es der CLI ermöglichen, NGRX-spezifischen Code zu generieren. Um z.B. NGRX-Unterstützung zum FlightBookingModule hinzuzufügen, genügt der folgende Aufruf:

```
ng g @ngrx/schematics:feature flight-booking/+state/flight-booking --module flight-booking\flight-booking.module.ts --creators --api false
```

Dieser Aufruf importiert StoreModule und EffectsModule in das FlightBookingModule und generiert die diskutierten Buildings-Blocks, wie z.B. Reducer oder Actions samt Testfällen (siehe Abbildung 17-4).



```
C:\Windows\System32\cmd.exe
>ng g @ngrx/schematics:feature flight-booking/+state/flight-booking --module flight-booking\flight-booking.module.ts --creators --api false
CREATE src/app/flight-booking/+state/flight-booking.actions.spec.ts (254 bytes)
CREATE src/app/flight-booking/+state/flight-booking.actions.ts (147 bytes)
CREATE src/app/flight-booking/+state/flight-booking.reducer.spec.ts (345 bytes)
CREATE src/app/flight-booking/+state/flight-booking.reducer.ts (376 bytes)
CREATE src/app/flight-booking/+state/flight-booking.effects.spec.ts (627 bytes)
CREATE src/app/flight-booking/+state/flight-booking.effects.ts (628 bytes)
CREATE src/app/flight-booking/+state/flight-booking.selectors.spec.ts (378 bytes)
CREATE src/app/flight-booking/+state/flight-booking.selectors.ts (268 bytes)
UPDATE src/app/flight-booking/flight-booking.module.ts (1678 bytes)
```

Abbildung 17-4: Store-Unterstützung für ein Feature-Module generieren

Das erste übergebene Argument legt fest, dass die Building-Blocks im Ordner *flight-booking* $\wedge$ *+state* zu generieren sind und das Präfix *flight-booking* erhalten. Das Plus (+) im Ordnernamen *+state* hat übrigens keine besondere Bedeutung. Es handelt sich vielmehr um eine übliche Konvention, die bewirkt, dass dieser Ordner in einer sortierten Ansicht als Erster erscheint.

Der Schalter --creators legt fest, dass die 2019 eingeführte Creators-API verwendet wird. Da sich diese mittlerweile zum Standard unter NGRX entwickelt hat, gehen wir hier ausschließlich darauf ein. Mit --api false legen wir fest, dass die CLI keine zusätzlichen Building-Blocks für die Kommunikation mit dem Backend generieren soll. Das übernehmen wir weiter unten, wenn wir genauer auf das Effects-Modul eingehen, selbst.

Neben diesen Dateien hat die CLI auch unser FlightBookingModule erweitert. Es importiert nun das StoreModule sowie das EffectsModule (siehe Beispiel 17-1).

*Beispiel 17-1: Das FlightBookingModule importiert jetzt das StoreModule und das EffectsModule.*

```
// src/app/flight-booking/flight-booking.module.ts
[...]
import { StoreModule } from '@ngrx/store';
import * as fromFlightBooking from './+state/flight-booking.reducer';
import { EffectsModule } from '@ngrx/effects';
import { FlightBookingEffects } from './+state/flight-booking.effects';

@NgModule({
  imports: [
    [...]
    StoreModule.forFeature(fromFlightBooking.flightBookingFeatureKey,
```

```

fromFlightBooking.reducer),
  EffectsModule.forFeature([FlightBookingEffects])
],
...
})
export class FlightBookingModule { }

```

Da es sich hier um ein Feature-Module handelt, kommt für den Import des Store Module und des EffectsModule die Methode `forFeature` zum Einsatz. Analog zur `forChild`-Methode des Routers verhindert hier der Einsatz von `forFeature`, dass die Anwendung zentrale Services in Feature-Modules erneut einrichtet und somit dupliziert oder überschreibt.

Diese Aufrufe konfigurieren die beiden Module auch mit von der CLI generierten Konstrukten:

*fromFlightBooking.flightBookingFeatureKey*

Konstante mit dem Namen des Zweigs, der im Zustandsbaum unser Feature-Module repräsentiert. Sie hat den Wert `flightBooking`.

*fromFlightBooking.reducer*

Grundgerüst eines Reducers, den wir gleich für unsere Zwecke anpassen werden.

*FlightBookingEffects*

Grundgerüst eines Effects, den wir gleich für unsere Zwecke anpassen werden.

## Building-Blocks implementieren

Nachdem Nx für die einzelnen Building-Blocks Dateien angelegt hat, gilt es, diese mit Leben zu erfüllen.

### State modellieren

Als Erstes stellt man sich dazu die Frage, wie sich der Zustand des Feature-Module gestaltet. Da wir Flüge laden wollen, liegt es nahe, im Zustand ein Array dafür vorzusehen. Außerdem möchten wir wissen, ob gerade Flüge geladen werden und ob gerade ein Fehler vorliegt.

Hierfür sehen die generierten Building-Blocks ein Interface State vor (siehe Beispiel 17-2).

*Beispiel 17-2: State-Interface für das FlightBookingModule*

```

// src/app/flight-booking/+state/flight-booking.reducer.ts

import { Action, createReducer, on } from '@ngrx/store';

// Hinzufügen:
import { Flight } from '../flight';
import * as FlightBookingActions from './flight-booking.actions';

```

```

export const flightBookingFeatureKey = 'flightBooking';

export interface State {
  // Hinzufügen:
  flights: Flight[];
  loading: boolean;
  error: unknown;
}

export const initialState: State = {
  // Hinzufügen:
  flights: [],
  loading: false,
  error: {}
};

[...]

```

Die Konstante `initialState` bekommt Standardwerte für die Einträge im State-Interface. Damit verhindert man die Werte `null` bzw. `undefined`, die üblicherweise zusätzliche Prüfungen notwendig machen.

Im Übrigen sind wir der Meinung, dass der Interface-Name zu allgemein ist. Deswegen benennen wir dieses Interface in `FlightBookingState` um:

```

// src/app/flight-booking/+state/flight-booking.reducer.ts

[...]

// Von State in FlightBookingState umbenennen:
export interface FlightBookingState {
  flights: Flight[];
  loading: boolean;
  error: string;
}

[...]

```

Das Umbenennen sollten Sie mit den Refactoring-Möglichkeiten Ihrer IDE durchführen, damit sie die Änderung in allen anderen Dateien nachzieht. In Visual Studio Code markieren Sie dazu den alten Namen und drücken dann `F2`.

Eine zentrale Idee von Redux ist, dass sich der gesamte Zustand in einem einzigen Zustandsbaum befindet (siehe Abbildung 17-3). Für diesen Zustandsbaum benötigen wir eine Wurzel, die auf den `FlightBookingState` sowie auf die Zustandsobjekte der anderen Module verweist.

Dieser Knoten hat also pro Feature-Module mindestens eine Eigenschaft. Um zu verhindern, dass man die Übersicht verliert, ist es üblich, pro Feature-Module eine eigene Sicht auf diesen sogenannten `AppState` einzurichten. Eine solche Sicht lässt sich durch ein Interface, das sich auf die im Feature-Module benötigten Eigenschaften beschränkt, ausdrücken (siehe Beispiel 17-3).

*Beispiel 17-3: AppState aus Sicht des FlightBookingModule*

```
// src/app/flight-booking/+state/flight-booking.reducer.ts

[...]

export const flightBookingFeatureKey = 'flightBooking';

// Hinzufügen:
export interface FlightBookingAppState {
  flightBooking: FlightBookingState;
}

// Nicht verändern
// (und nicht mit dem FlightBookingAppState verwechseln):
export interface FlightBookingState {
  flights: Flight[];
  loading: boolean;
  error: string;
}
```

Auch wenn dieses Interface unsere Anforderungen erfüllt, gibt es hier eine kleine Unschönheit: Der Name `flightBooking` befindet sich sowohl in der Konstanten `flightBookingFeatureKey` als auch im neuen Interface `FlightBookingAppState`. Die Konstante kommt, wie weiter oben besprochen, beim Registrieren des `StoreModule` zum Einsatz. Sie beinhaltet genauso wie das neue Interface den Namen jenes Zweigs im Zustandsbaum, den unser `FlightBookingModule` verwendet.

Bei einer Namensänderung müssen wir also immer diese beiden Stellen anpassen. Zum Glück gibt uns TypeScript die Möglichkeit, eine Eigenschaft nach einer Konstanten zu benennen (siehe Beispiel 17-4).

*Beispiel 17-4: Eigenschaft in FlightBookingAppState nach der Konstanten `flightBookingFeatureKey` benennen*

```
// src/app/flight-booking/+state/flight-booking.reducer.ts

[...]

export const flightBookingFeatureKey = 'flightBooking';

export interface FlightBookingAppState {
  // Verweis auf Konstante in eckigen Klammern:
  [flightBookingFeatureKey]: FlightBookingState;
}

export interface FlightBookingState {
  flights: Flight[];
  loading: boolean;
  error: string;
}
```

## Actions festlegen

Als Nächstes stellt man sich die Frage, welche Aktionen mit dem State auszuführen sind. Alle diese Aktionen werden in der Datei `flight-booking.action.ts` eingerichtet. Auch hier können Sie die von der CLI generierten Konstrukte mittels Refactoring anpassen (siehe Beispiel 17-5).

*Beispiel 17-5: Actions für das FlightBookingModule*

```
// src/app/flight-booking/+state/flight-booking.actions.ts

import { createAction, props } from '@ngrx/store';
import { Flight } from '../flight';

export const loadFlights = createAction(
  '[FlightBooking] loadFlights',
  props<{from: string; to: string}>()
);

export const flightsLoaded = createAction(
  '[FlightBooking] flightsLoaded',
  props<{flights: Flight[]}>()
);

export const loadFlightsError = createAction(
  '[FlightBooking] loadFlightsError',
  props<{error: string}>()
);
```

Die zugewiesenen Namen müssen eindeutig sein, da NGRX die Aktionen daran unterscheiden kann. Es gehört auch zum guten Ton, dem Namen eine Kategorie – hier `[FlightBooking]` – voranzustellen. Diese Kategorie soll beim Debuggen über die Herkunft der Action, z.B. das Modul, aus dem sie stammt, informieren.

Gerade asynchrone Operationen wie das Laden von Flügen gilt es, durch mehrere Actions zu repräsentieren. Beispielsweise fordert `loadFlights` das Laden von Flügen an, während `flightsLoaded` anzeigt, dass dieser Vorgang erfolgreich war, und dem Store die ermittelten Flüge übergibt. Mit `loadFlightsError` weist die Anwendung hingegen auf einen Fehler beim Laden hin.

## Reducer definieren

Der Reducer, der sich um das Abarbeiten der Actions kümmert, findet sich in der Datei `flight-booking.reducer.ts`.

*Beispiel 17-6: Reducer für das FlightBookingModule*

```
// src/app/flight-booking/+state/flight-booking.reducer.ts

[...]

export const reducer = createReducer(
```

```

initialState,
on(FlightBookingActions.flightsLoaded, (state, action) => {
  const flights = action.flights;
  const loading = false;
  return {...state, flights, loading};
}),
on(FlightBookingActions.loadFlights, (state, action) => {
  const flights = [] as Flight[];
  const loading = false;
  return {...state, flights, loading};
}),
on(FlightBookingActions.loadFlightsError, (state, action) => {
  const flights = [] as Flight[];
  const loading = false;
  const error = action.error;
  return {...state, flights, loading, error};
}),
);

```

Jeder im Reducer mit `on` eingerichtete Handler nimmt den aktuellen State sowie die aktuelle Action entgegen und liefert einen neuen State zurück. Wichtig ist, dass der State nicht verändert werden darf – bei Redux ist er per definitionem immutable. Stattdessen erzeugt der Reducer jeweils ein neues State-Objekt. Dazu klonnt er mit dem Spread-Operator das alte Objekt und tauscht dabei die gewünschten Eigenschaften aus (siehe Kapitel 1).

Beim Handler für `loadFlights` fällt auf, dass lediglich das Array mit dem Suchergebnis zurückgesetzt wird. Da Reducer immer synchron arbeiten, findet das asynchrone Laden außerhalb des Stores in einem Effect statt.

## Effect einrichten

Der Effect für das Laden von Flügen kommt in die generierte Datei `flight-booking.effects.ts` (siehe Beispiel 17-7).

*Beispiel 17-7: Effect für das FlightBookingModule*

```

// src/app/flight-booking/+state/flight-booking.effects.ts

import { Injectable } from '@angular/core';
import { Actions, createEffect, ofType } from '@ngrx/effects';
import { of } from 'rxjs';

import { catchError, map, switchMap } from 'rxjs/operators';
import { FlightService } from '../flight.service';
import { flightsLoaded, loadFlights, loadFlightsError } from './flight-booking.actions';

@Injectable()
export class FlightBookingEffects {

```

```

flightsLoad$ = createEffect(() => this.actions$.pipe(
  ofType(loadFlights),
  switchMap(a => this.flightService.find(a.from, a.to).pipe(
    map(flights => flightsLoaded({flights})),
    catchError(error => of(loadFlightsError({error})))
  )),
));

constructor(
  private flightService: FlightService,
  private actions$: Actions) {}

}

```

Da der Effect technisch gesehen ein Service ist, kann er sich andere Services injizieren lassen. Auf diese Weise erhält er unseren FlightService sowie ein Actions-Objekt. Dabei handelt es sich um ein Observable, das sämtliche Aktionen, die auch an den Store gesendet werden, empfängt. Daraus gilt es nun weitere Observables, die sich um die Ausführung der Effects kümmern, abzuleiten.

Da wir uns hier nur für loadFlights interessieren, filtert der Effect mit ofType die empfangenen Actions. Der Operator switchMap delegiert an den FlightService. Der Einsatz von switchMap ist hier notwendig, weil find ein neues Observable liefert, auf das zu wechseln ist. Dieses transportiert ein Flight-Array, das map in einer flights Loaded-Action verpackt. Diese Action sendet @ngrx/effects an den Store, wo sich der gezeigte Reducer darum kümmert.

Im Fehlerfall bildet catchError den erhaltenen Fehler auf eine loadFlightsError-Action ab. Auch dieser wird von unserem Reducer im Store verarbeitet.

## Auf den Store zugreifen

Die gute Nachricht vorweg: Den schwierigen Teil haben wir hinter uns gebracht. Jetzt müssen die einzelnen Komponenten den Store nur noch konsumieren. Zur Demonstration injizieren wir den Store in die FlightSearchComponent (siehe Beispiel 17-8).

*Beispiel 17-8: Die FlightSearchComponent nutzt nun den Store.*

```

// src/app/flight-search/flight-search.component.ts

[...]

// Hinzufügen:
import { Store } from '@ngrx/store';
import { loadFlights } from '../+state/flight-booking.actions';
import { FlightBookingAppState, flightBookingFeatureKey }
  from '../+state/flight-booking.reducer';

@Component({ ... })
export class FlightSearchComponent implements OnInit {

```

```

[...]

// Hinzufügen bzw. anpassen:
flights$ = this.store.select(appState => appState[flightBookingFeatureKey].flights);

[...]

constructor(
  // Hinzufügen:
  private store: Store<FlightBookingAppState>,
  private flightService: FlightService) {
}

ngOnInit(): void { }

// Anpassen:
search(): void {
  this.store.dispatch(loadFlights({from: this.from, to: this.to}));
}

[...]
}

```

Der Store ist mit unserer Sicht auf die Wurzel des Zustandsbaums zu typisieren. Unsere `FlightSearchComponent` bezieht nun das Observable `flights$` mit den geladenen Flügen aus dem Store. Dazu verwendet sie die Methode `select`.

Um das Laden anzustoßen, versendet die Methode `search` die Action `loadFlights`. Hierzu kommt die Methode `dispatch` des Stores zum Einsatz. Die Methoden `select` und `dispatch` sind tatsächlich die einzigen, die wir vom Store benötigen.

Würden wir die gesamte Komponente auf den Store umstellen, könnten wir auch den Verweis auf den `FlightService` entfernen. Diese Vorgehensweise vereinfacht die einzelnen Komponenten, da diese nur mehr mit dem Store interagieren müssen.

## Debuggen mit dem Store

Das Chrome-Plug-in *Redux DevTools* ([tinyurl.com/chrome-redux](http://tinyurl.com/chrome-redux)) hilft erheblich beim Debuggen von Lösungen, die auf NGRX basieren. Unter anderem zeigt es den aktuellen Inhalt des Stores und die verwendeten Actions. Um unsere Lösung damit zu verbinden, benötigen wir das Paket `@ngrx/store-devtools`:

```
npm install @ngrx/store-devtools
```

Danach müssen Sie gegebenenfalls Ihre IDE, z.B. Visual Studio Code, neu starten. Um dieses Paket zu nutzen, ist das `StoreDevtoolsModule` ins `AppModule` zu importieren (siehe Beispiel 17-9).

*Beispiel 17-9: StoreDevtoolsModule ins AppModule installieren*

```
// src/app/app.module.ts
```

```
[...]
```

```

// Hinzufügen:
import { StoreDevtoolsModule } from '@ngrx/store-devtools';
import { environment } from 'src/environments/environment';

@NgModule({
  imports: [
    ...
  ]

  // Hinzufügen:
  !environment.production ? StoreDevtoolsModule.instrument() : []
],
...
})
export class AppModule { }

```

Üblicherweise kommt das `StoreDevtoolsModule` nur im Debug-Modus zum Einsatz. Im Produktivmodus wäre der dadurch verursachte Overhead zu groß.



Starten Sie nach dem Installieren der Redux DevTools ([tinyurl.com/chrome-redux](http://tinyurl.com/chrome-redux)) Ihren Browser neu.

Wenn Sie nun Ihre Anwendung starten, finden Sie in den Chrome DevTools ein Registerblatt *Redux* (siehe Abbildung 17-5).

Abbildung 17-5: Redux DevTools im Browser

Hier können Sie die einzelnen Actions sowie die dadurch ausgelösten Änderungen am Zustandsbaum und den aktuellen Anwendungszustand einsehen. Außerdem können Sie unter anderem mit dem Schieberegler am unteren Ende in der Zeit zurückgehen, um einen früheren Anwendungszustand herbeizuführen. Sie können auch die einzelnen Zustandsänderungen noch mal abspielen, um sie besser nachzuvollziehen.

# Selektoren

In der zuvor beschriebenen Umsetzung greift die FlightSearchComponent direkt auf den Store zu:

```
flights$ = this.store.select  
  (appState => appState[flightBookingFeatureKey].flights);
```

Das ist zwar einfach, hat aber auch den Nachteil, dass die Komponente an den Aufbau des Stores gebunden wird. Dies erschwert ein nachträgliches Refactoring. Selektoren lösen das Problem, indem sie die nötigen Zugriffspfade an zentrale Stellen auslagern. Außerdem können Selektoren eventuelle Transformationen, die sie auf abgerufene Daten durchführen, cachen. Das verbessert die Performance.

## Ein erster Selektor

Um den Einsatz von Selektoren zu demonstrieren, erweitern wir zunächst den FlightBookingState um eine Eigenschaft hide (siehe Beispiel 17-10).

*Beispiel 17-10: FlightBookingState mit Eigenschaft hide*

```
// src/app/flight-booking/+state/flight-booking.reducer.ts  
[...]  
  
export interface FlightBookingState {  
  flights: Flight[];  
  loading: boolean;  
  error: unknown;  
  // Hinzufügen:  
  hide: number[];  
}  
  
export const initialState: FlightBookingState = {  
  flights: [],  
  loading: false,  
  error: {},  
  // Hinzufügen:  
  hide: [4]  
};
```

Diese Eigenschaft soll die IDs von Flügen aufnehmen, die die Anwendung nicht anzeigen soll. Damit die Komponenten weder diese Logik noch den Aufbau des Zustandsbaums kennen müssen, richten wir dafür einen Selektor ein. Die CLI hat dafür bereits die Datei *flight-booking.selectors.ts* eingerichtet, die sich für unsere Zwecke erweitern lässt (siehe Beispiel 17-11).

*Beispiel 17-11: Selektoren für das FlightBookingModule*

```
// src/app/flight-booking/+state/flight-booking.selectors.ts  
[...]
```

```

// Hinzufügen:
import { createSelector } from '@ngrx/store';
import { flightBookingFeatureKey } from './flight-booking.reducer';

[...]

// Hinzufügen:
export const selectFlights = createSelector(
  (appState: FlightBookingAppState) => appState[flightBookingFeatureKey].flights,
  (appState: FlightBookingAppState) => appState[flightBookingFeatureKey].hide,
  (flights, hide) => flights.filter(f => !hide.includes(f.id))
);

```

Die ersten beiden an `createSelector` übergebenen Lambda-Ausdrücke rufen zunächst die Eigenschaften `flights` und `hide` aus dem Zustandsbaum ab. Diese Werte übergibt `createSelector` an den letzten Lambda-Ausdruck. Dabei handelt es sich um einen sogenannten *Projector*, der die abgerufenen Werte auf das gewünschte Ergebnis abbildet. Diesen Wert cacht der Selektor so lange, bis sich die zugrunde liegenden Werte ändern.



Von `createSelector` existieren übrigens mehrere Überladungen, so dass sie bis zu acht Lambda-Ausdrücke, die Werte aus dem Zustandsbaum abrufen, übergeben können.

Dieser Selektor lässt sich jetzt in der `FlightSearchComponent` nutzen (siehe Beispiel 17-12).

*Beispiel 17-12: Die FlightSearchComponent greift nun über einen Selektor auf die Flüge zu.*

```

// src/app/flight-search/flight-search.component.ts
[...]

// Hinzufügen:
import { selectFlights } from '../state/flight-booking.selectors';

[...]

export class FlightSearchComponent implements OnInit {

  [...]

  // Aktualisieren:
  flights$ = this.store.select(selectFlights);

  [...]
}

```

## Selektoren verschachteln

Um Wiederholungen zu vermeiden, können Selektoren auch Werte anderer Selektoren nutzen (siehe Beispiel 17-13).

### Beispiel 17-13: Verschachtelte Selektoren

```
// src/app/flight-booking/+state/flight-booking.selectors.ts  
[...]  
  
// Auf Knoten flightBooking (= Wert von fromFlightBooking.flightBookingFeatureKey)  
// zugreifen:  
export const selectFlightBookingState =  
  createFeatureSelector<fromFlightBooking.FlightBookingState>(  
    fromFlightBooking.flightBookingFeatureKey  
  );  
  
export const selectAllFlights = createSelector(  
  selectFlightBookingState,  
  fbs => fbs.flights  
);  
  
export const selectFlightsToHide = createSelector(  
  selectFlightBookingState,  
  fbs => fbs.hide  
);  
  
export const selectFilteredFlights = createSelector(  
  selectAllFlights,  
  selectFlightsToHide,  
  (flights, hide) => flights.filter(f => !hide.includes(f.id))  
);
```

In diesem Beispiel kommt auch ein sogenannter Feature-Selektor zum Einsatz. Dabei handelt es sich um einen Selektor, der die Wurzel des Zustandsbaums auf eine ihrer Eigenschaften abbildet. Der Name dieser Eigenschaft ist als String zu übergeben. Das gezeigte Beispiel übergibt beispielsweise die Konstante `fromFlightBooking.flightBookingFeatureKey` mit dem Wert `flightBooking`. Deswegen fördert dieser Feature-Selektor den `FlightBookingState` zutage.

## Meta-Reducer

Logiken, die Sie in vielen oder allen Reducern stattfinden lassen möchten, können Sie in sogenannte Meta-Reducer auslagern. Das sind Reducer, die die herkömmlichen Reducer ummanteln. In der Regel delegieren sie an die herkömmlichen Reducer. Davor und danach können sie jedoch eigene Logiken anstoßen.

Beispiel 17-14 zeigt ein Beispiel, das sämtliche Zustände und Actions vor dem Ausführen der adressierten Reducer protokolliert.

### Beispiel 17-14: Ein Meta-Reducer zum Protokollieren von Zustandsänderungen

```
// src/app/+state/meta.reducer.ts  
  
import { ActionReducer } from '@ngrx/store';  
  
export function debug(reducer: ActionReducer<any>): ActionReducer<any> {
```

```

        return function(state, action) {
            // Protokollieren:
            console.log('state', state);
            console.log('action', action);

            // An "herkömmlichen" Reducer delegieren:
            return reducer(state, action);
        };
    }
}

```

Damit die Anwendung Ihre Meta-Reducer ausführt, sind sie beim Aufruf von Store Module.`forRoot` im AppModule zu registrieren (siehe Beispiel 17-15).

*Beispiel 17-15: Registrierung eines Meta-Reducers beim AppModule*

```

// src/app/app.module.ts

[...]
// Hinzufügen:
import { debug } from './+state/meta.reducer';

@NgModule({
    imports: [
        [...]
        StoreModule.forRoot({}, {
            // Meta-Reducer registrieren:
            metaReducers: [debug]
        }),
        [...]
    ],
    ...
})
export class AppModule { }

```

## Zusammenfassung

Die Nutzung von `@ngrx/store` und `@ngrx/effects` zwingt uns ein strukturiertes Vorgehen auf. Ausgehend vom Zustandsbaum über Actions und Reducer bzw. Effects handelt man sich zu den Komponenten vor. Redux kommt mit einer sehr genauen Vorstellung davon, wie die »Welt« funktionieren hat, und lenkt daher das Entwicklungsteam. Es gibt also wenige Diskussionen darüber, wo etwas hingehört. Das trifft sicher nicht den Geschmack aller Entwicklerinnen und Entwickler, erweist sich bei größeren Teams jedoch als vorteilhaft.

Zyklen vermeidet Redux, indem es alle Aktionen an jeden Reducer sendet. Somit können die unterschiedlichen Anwendungsteile in einem Atemzug auf ein Ereignis reagieren. Ereigniskaskaden sind somit weitgehend unnötig. Trotz eines zentralen Zustands sind die einzelnen Actions, Reducer und Effects unabhängig voneinander und lassen sich in jeweils eigenen Dateien verstauen.

Mit den richtigen Werkzeugen verbessert sich sogar die Nachvollziehbarkeit: Kommt der Store durchgängig zum Einsatz, ergibt sich der aktuelle Anwendungszustand aus der Folge der einzelnen Aktionen. Protokolliert die Anwendung diese, lässt sich leicht herausfinden, wie der Benutzer einen bestimmten Fehlerzustand herbeigeführt hat. Unterstützend greifen sogenannte zeitreisende Debugger ein, die ausgewählte Aktionen rückgängig machen und es erlauben, die durchgeföhrten Aktionen nochmals abzuspielen. Ein Beispiel dafür sind die Redux DevTools, die als Chrome-Plug-in zur Verfügung stehen.

Da `@ngrx/store` und `@ngrx/effects` mit Angular im Hinterkopf entworfen wurden, passen sie auch wunderbar in die Welt von Angular und berücksichtigen dort vorherrschende Konzepte wie Module, Lazy Loading und Services. Da Immutables und Observables konsequent zum Einsatz kommen, lässt sich bei den Komponenten der Optimierungsmodus OnPush aktivieren. Dieser führt zu einer stark verbesserten Datenbindungsperformance.

Wo so viel Licht ist, gibt es natürlich auch Schatten. Ein Nachteil ist die gewöhnungsbedürftige Arbeit mit Immutables. Ein anderer Nachteil besteht darin, dass die zusätzlichen Building-Blocks die Komplexität der Anwendung erhöhen. Glücklicherweise gibt es Codegeneratoren wie Nx, die diesen Umstand kompensieren.

# Details zu Komponenten und Direktiven

Angular ist durch und durch komponentenorientiert. Aus diesem Grund kommen im vorliegenden Werk Komponenten seit dem ersten Angular-Beispiel immer wieder vor. In diesem Kapitel knüpfen wir daran an und präsentieren einige weiterführende Aspekte, die vor allem bei der Schaffung wiederverwendbarer und anpassbarer Steuerelemente nützlich sind.

Zusätzlich werfen wir einen genaueren Blick auf das Konzept der Direktiven, das mit jenem der Komponenten verwandt ist. Wie die kommenden Abschnitte zeigen werden, handelt es sich bei Komponenten genau genommen lediglich um eine spezielle Ausprägung von Direktiven.

Die Informationen in diesem Kapitel werden für Sie besonders dann interessant sein, wenn Sie eine wiederverwendbare Komponentenbibliothek schaffen wollen. Außerdem hilft Ihnen dieses Kapitel dabei, das Verhalten von Angular besser zu verstehen.

## Vorbereitungen

Für die Beispiele in diesem Kapitel nutzen wir ein neues Angular-Modul, dessen Komponenten Informationen für Kunden anbieten. Das Modul heißt `CustomerModule`, und seine erste Komponente ist die `BookingHistoryComponent`. Dafür richten wir auch eine Route und einen Menüeintrag ein.

Die nachfolgenden Schritte leiten Sie durch diese Aufgaben. Wir starten mit dem Generieren des Moduls:

```
ng g module customer
```

Eine erste Komponente wird sich um das Anzeigen von Flugbuchungen kümmern:

```
ng g c customer/booking-history
```

Damit sich die Komponenten des neuen Moduls über den Router aktivieren lassen, erhält es eine Routenkonfiguration:

```
// src/app/customer/customer.routes.ts

import { Routes } from '@angular/router';
import { BookingHistoryComponent } from './booking-history/booking-history.
component';

export const CUSTOMER_ROUTES: Routes = [
{
    path: 'customer/booking-history',
    component: BookingHistoryComponent
}
];
```

Die Routenkonfiguration ist samt dem RouterModule ins neue CustomerModule zu importieren. Dasselbe gilt für unser SharedModule (siehe Beispiel 18-1).

*Beispiel 18-1: Das CustomerModule importiert die neue Routenkonfiguration.*

```
// src/app/customer/customer.module.ts

import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

// Importe hinzufügen:
import { RouterModule } from '@angular/router';
import { CUSTOMER_ROUTES } from './customer.routes';
import { SharedModule } from '../shared/shared.module';
import { BookingHistoryComponent } from './booking-history/booking-history.component';

@NgModule({
  imports: [
    CommonModule,
    // Shared-Modul einfügen:
    SharedModule,
    // Routenkonfiguration einfügen:
    RouterModule.forChild(CUSTOMER_ROUTES)
  ],
  declarations: [BookingHistoryComponent]
})
export class CustomerModule { }
```

Bitte beachten Sie, dass wir hier die Reihenfolge von imports und declarations gegenüber der generierten Reihenfolge geändert haben.

Zusätzlich ist das CustomerModule ins AppModule zu importieren (siehe Beispiel 18-2).

*Beispiel 18-2: Das AppModule importiert das neue CustomerModule.*

```
// src/app/app.module.ts

[...]
// Import hinzufügen:
import { CustomerModule } from './customer/customer.module';
```

```

@NgModule({
  imports: [
    [...]
    // Modul registrieren:
    CustomerModule
  ],
  declarations: [
    [...]
  ],
  providers: [],
  bootstrap: [
    AppComponent
  ]
})
export class AppModule { }

```

Außerdem erzeugen wir einen Hauptmenüpunkt im Template der SidebarComponent:

```

<!-- src/app/sidebar/sidebar.component.html -->
[...]

<li routerLinkActive="active">
  <a routerLink="customer/booking-history">
    <p>Booking History</p>
  </a>
</li>
[...]

```

Starten wir nun die Anwendung, sollte der neue Menüpunkt *Booking History* zur zuvor generierten *BookingHistoryComponent* führen (siehe Abbildung 18-1).

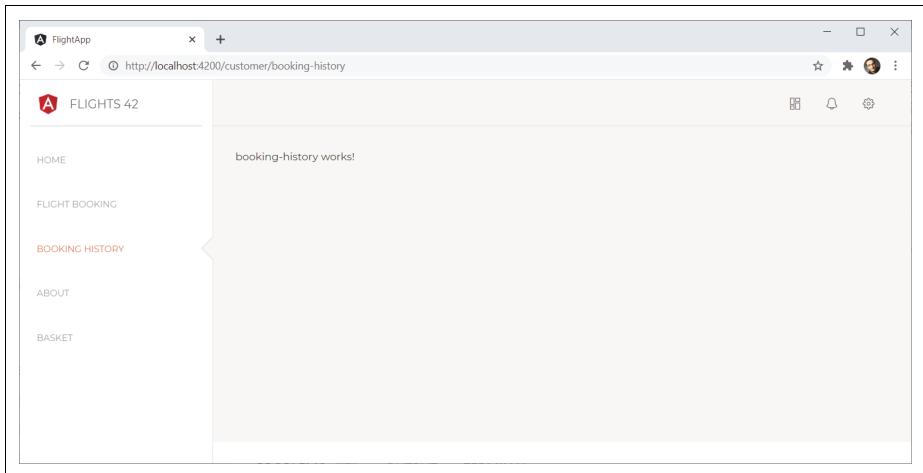


Abbildung 18-1: Die neue Route führt zur *BookingHistoryComponent*.

# Weiterführende Aspekte von Komponenten

Um weiterführende Aspekte von Komponenten zu veranschaulichen, kommt in diesem Abschnitt ein einfaches Registerblatt-Steuerelement zum Einsatz (siehe Abbildung 18-2).

The screenshot shows a register card with three tabs at the top: 'Upcoming Flights' (highlighted in red), 'Operated Flights', and 'Cancelled Flights'. The main section is titled 'Upcoming Flights' and contains a table with three rows of flight information:

1	Hamburg	Berlin	01.02.2025 17:00
2	Hamburg	Frankfurt	01.02.2025 17:30
3	Hamburg	Mallorca	01.02.2025 17:45

Abbildung 18-2: Registerblatt als Fallbeispiel für dieses Kapitel

Die Links im oberen Bereich ermöglichen es, zwischen den einzelnen Registerblättern umzuschalten. Das gesamte Registerblatt-Steuerelement repräsentiert die Implementierung durch eine Komponente, die im Folgenden als TabbedPane bezeichnet wird, und für die einzelnen Registerblätter kommt jeweils eine Tab-Komponente zum Einsatz.

Sie können diese mit der CLI oder mit dem Plug-in *Angular Schematics* in Visual Studio generieren:

```
ng g c shared/controls/tabbed-pane --export  
ng g c shared/controls/tab --export
```

Vergewissern Sie sich nach dem Generieren, dass das SharedModule die beiden neuen Komponenten sowohl deklariert als auch exportiert.

## Content Projection

*Content Projection* ermöglicht einer Komponente, vom Aufrufer Markup – zum Beispiel HTML – entgegenzunehmen. Diesen kann es im eigenen Template anzeigen.

Unsere TabComponent benötigt dieses Verhalten, da sie die übergebenen Elemente innerhalb des von ihr angezeigten Tabs präsentieren muss:

```
<app-tab title="Upcoming Flights">  
  <p>No upcoming flights!</p>  
</app-tab>
```

Um zu zeigen, wie das möglich ist, werfen wir einen Blick auf die Implementierung der Komponente. Beispiel 18-3 zeigt die Komponentenklasse.

*Beispiel 18-3: Implementierung der TabComponent*

```
// src/app/shared/controls/tab/tab.component.ts

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

@Component({
  selector: 'app-tab',
  templateUrl: './tab.component.html',
  styleUrls: ['./tab.component.scss']
})
export class TabComponent implements OnInit {

  @Input() title = '';
  visible = true;

  ngOnInit(): void {
  }

}
```

Die Implementierung beschränkt sich an dieser Stelle auf eine Eigenschaft für den Titel und einer weiteren Eigenschaft visible, aus der hervorgeht, ob das Registerblatt angezeigt werden soll. Der Titel lässt sich über die Datenbindung festlegen (Input).

Das Template bindet sich an diese beiden Eigenschaften (siehe Beispiel 18-4).

*Beispiel 18-4: Template der TabComponent mit Platzhalter*

```
<!-- src/app/shared/controls/tab/tab.component.html -->

<div *ngIf="visible">
  <h2>{{title}}</h2>
  <ng-content></ng-content>
</div>
```

Um festzulegen, wo der Titel innerhalb des Registerblatts einzublenden ist, markiert die Komponente die gewünschte Stelle mit dem Element ng-content.



Eine Komponente kann auch mehrere Platzhalter mit ng-content definieren. In diesem Fall erhält jedes ng-content einen CSS-Selektor, der die anzuseigenden Teile des übergebenen Markups adressiert. Im folgenden Beispiel verweisen die Selektoren auf die Elemente mit den Klassen header und content:

```
<ng-content select=".header"></ng-content>
<ng-content select=".content"></ng-content>
```

Beim Aufruf der Komponente sind dann entsprechende Elemente zu übergeben:

```
<app-tab title="Paid">
  <div class="header">
    Header
  </div>
  <div class="content">
```

```
    Content
  </div>
</app-tab>
```

Um die neue Komponente auszuprobieren, können wir sie im Template der BookingHistoryComponent aufrufen (siehe Beispiel 18-5).

*Beispiel 18-5: Einsatz der TabComponent in der BookingHistoryComponent*

```
<!-- src/app/customer/booking-history/booking-history.component.html -->

<h1>Booking History</h1>

<app-tab title="Upcoming Flights">
  <p>No upcoming flights!</p>
</app-tab>

<app-tab title="Operated Flights">
  <p>No operated flights!</p>
</app-tab>

<app-tab title="Cancelled Flights">
  <p>No cancelled flights!</p>
</app-tab>
```

Das Ergebnis zeigt die drei Registerblätter einfach untereinander an.

Damit jeweils nur das aktive Registerblatt angezeigt wird, bringen wir im nächsten Abschnitt die TabbedPaneComponent ins Spiel. Sie gruppiert die Registerblätter und blendet sie ein bzw. aus.

## Parent-Komponenten referenzieren

Die TabbedPaneComponent soll beim Aufruf über Content Projection die anzuseigenenden TabComponents übergeben bekommen:

```
<app-tabbed-pane>
  <app-tab title="Upcoming Flights">
    <p>No upcoming flights!</p>
  </app-tab>

  <app-tab title="Operated Flights">
    <p>No operated flights!</p>
  </app-tab>

  <app-tab title="Cancelled Flights">
    <p>No cancelled flights!</p>
  </app-tab>
</app-tabbed-pane>
```

Diese Registerblätter verwaltet sie in ihrer Eigenschaft tabs (siehe Beispiel 18-6).

*Beispiel 18-6: Implementierung der TabbedPaneComponent*

```
// src/app/shared/controls/tabbet-pane/tabbet-pane.component.ts

import { AfterContentInit, Component, OnInit } from '@angular/core';
import { TabComponent } from '../tab/tab.component';

@Component({
  selector: 'app-tabbet-pane',
  templateUrl: './tabbed-pane.component.html',
  styleUrls: ['./tabbed-pane.component.scss']
})
export class TabbedPaneComponent implements OnInit, AfterContentInit {

  tabs: Array<TabComponent> = [];
  activeTab: TabComponent | undefined;

  constructor() { }

  ngAfterContentInit(): void {
    if (this.tabs.length > 0) {
      this.activate(this.tabs[0]);
    }
  }

  ngOnInit(): void {
  }

  register(tab: TabComponent): void {
    this.tabs.push(tab);
  }

  activate(active: TabComponent): void {
    for (const tab of this.tabs) {
      tab.visible = (tab === active);
    }
    this.activeTab = active;
  }
}
```

Die Eigenschaft `activeTab` repräsentiert das gerade angezeigte Registerblatt. Diese Implementierung geht davon aus, dass sich alle Registerblätter bei der TabbedPane Component mit der Methode `register` anmelden. Um diese Aufgabe kümmern wir uns weiter unten.

Die Komponente verwendet neben dem üblichen `OnInit`-Life-Cycle-Hook auch den Hook `AfterContentInit`. Seine `ngAfterContentInit`-Methode führt Angular aus, sobald der übergebene Content initialisiert wurde. In unserem Fall sind das die einzelnen Registerblätter.

Die Implementierung von `ngAfterContentInit` nutzt die `active`-Methode, um das erste Registerblatt einzublenden bzw. alle anderen auszublenden.

Um den Navigationsbereich zum Umschalten zwischen den Registerblättern anzuzeigen, iteriert das Template der TabbedPaneComponent die einzelnen Registerblätter und rendert für jedes einen Link (siehe Beispiel 18-7).

*Beispiel 18-7: Template der TabbedPaneComponent*

```
<!-- src/app/shared/controls/tabbed-pane/tabbed-pane.component.html -->

<div class="tabbed-pane">

    <div class="navigation">
        <span *ngFor="let tab of tabs" class="tab-link">
            <a [ngClass]="{active: tab == activeTab}" (click)="activate(tab)">
                {{tab.title}}</a>
        </span>
    </div>

    <ng-content></ng-content>

</div>
```

Das Template nutzt ng-content, um die gruppierten Registerblätter einzublenden. Durch Content Projection platziert Angular somit sämtliche Registerblätter in diesem Element. Durch die oben beschriebenen Mechanismen, allen voran jenen in der Methode activate, zeigt sich jedoch zeitgleich nur eines dieser Registerblätter.

Zum Styling der TabbedPaneComponent nutzen wir die folgenden Einträge in ihrem Stylesheet.

*Beispiel 18-8: Styles der TabbedPaneComponent*

```
/* src/app/shared/controls/tabbed-pane/tabbed-pane.component.scss */

.navigation {
    margin-bottom: 30px;
}

.tab-link {
    font-size: 16px;
    padding-bottom: 3px;
    border-bottom: 5px solid darkseagreen;
    margin-right: 10px;
}

.tab-link a {
    color: black;
    cursor: pointer;
}

.tab-link a:hover {
    color: orangered;
    text-decoration: none;
}
```

```
.tab-link a.active {  
    color: orangered;  
}
```

Nun müssen wir nur noch die TabComponent dazu bringen, sich bei ihrer TabbedPaneComponent anzumelden. Dazu braucht die TabComponent zunächst eine Referenz auf die TabbedPaneComponent. Glücklicherweise stellt Angular sämtliche im DOM übergeordneten Komponenten (Parent, Parent des Parents etc.) über Dependency Injection zur Verfügung. Somit kann sich die TabComponent ihre TabbedPaneComponent in den Konstruktor injizieren lassen (siehe Beispiel 18-9).

*Beispiel 18-9: Parent-Komponente in Child-Komponente injizieren*

```
// src/app/shared/controls/tab/tab.component.ts  
  
import { Component, Input, OnInit } from '@angular/core';  
  
//Hinzufügen:  
import { TabbedPaneComponent } from '../tabbed-pane/tabbed-pane.component';  
  
@Component({  
    selector: 'app-tab',  
    templateUrl: './tab.component.html',  
    styleUrls: ['./tab.component.scss']  
})  
export class TabComponent implements OnInit {  
  
    @Input() title = '';  
    visible = true;  
  
    //Hinzufügen bzw. anpassen:  
    constructor(pane: TabbedPaneComponent) {  
        pane.register(this);  
    }  
  
    ngOnInit(): void {  
    }  
}
```

Um sich beim injizierten Parent anzumelden, übergibt sich die TabComponent selbst (`this`) an dessen `register`-Methode.



Wollte man sicherstellen, dass die TabComponent auch ohne übergeordnete TabbedPaneComponent funktioniert, könnte der Konstruktor das Argument mit `Optional` dekorieren:

```
constructor(@Optional() pane: TabbedPaneComponent) {  
    if (pane) {  
        pane.register(this);  
    }  
}
```

Kann Angular in diesem Fall für das Argument keine Implementierung finden, übergibt es lediglich `null`.

## View und Content

Jede Komponente kann neben einer View auch einen sogenannten Content aufweisen. Da die Unterscheidung dieser beiden Konzepte nicht ganz einfach ist, zeigt Abbildung 18-3 den Unterschied anhand der zuvor betrachteten TabComponent.

Das Diagramm zeigt den Quellcode einer TabComponent. Die View ist als grüner Bereich markiert, der das Template-Element umfasst. Der Content ist als blauer Bereich markiert, der das ng-content-Element umfasst. Ein Pfeil weist von der View auf das Content-Element.

```
@Component({
  selector: 'app-tab',
  template: `
    <div *ngIf="visible">
      <h2>{{title}}</h2>
      <ng-content></ng-content>
    </div>
  `,
  styleUrls: ['./tab.component.scss']
})
export class TabComponent {
  @Input() title = '';
  visible = true;
}
```

Abbildung 18-3: Unterschied zwischen View und Content

Der Übersicht halber wurden hier die Templates direkt als Strings in die Komponenten aufgenommen.

Während die View durch das Template definiert wird und somit die Präsentation der Komponente zur Laufzeit bestimmt, handelt es sich beim Content um das Markup, das der Aufrufer an die Komponente übergibt. Wie wir weiter oben schon erwähnt haben, platziert Angular mittels Content Projection dieses Markup im ng-content-Element des Templates.

### Interaktion mit dem Content

Eine Komponente kann direkt auf ihre View und ihren Content zugreifen und einzelne darin platzierte Komponenten abfragen. Um den Zugriff auf den Content zu veranschaulichen, ändern wir unsere TabbedPaneComponent und die dazugehörige TabComponent ein wenig ab.

Bisher hat sich die TabComponent bei ihrem Parent, der TabbedPaneComponent, registriert. Das ist möglich, weil sich jede Komponente alle ihr im DOM übergeordneten Komponenten (Vorfahren) injizieren lassen kann. Untergeordnete Komponenten bekommen somit Zugriff auf übergeordnete Komponenten.

Dieses Spiel wollen wir hier umdrehen: Die TabbedPaneComponent soll nun Kontakt mit ihren TabComponents aufnehmen. Dazu muss sie auf ihren Content zugreifen (siehe Beispiel 18-10).

*Beispiel 18-10: Mit ContentChildren den Content abfragen*

```
// src/app/shared/controls/tabbet-pane/tabbet-pane.component.ts
```

```
import { AfterContentInit, Component, ContentChildren, OnInit, QueryList }
```

```

from '@angular/core';
import { TabComponent } from '../tab/tab.component';

@Component({
  selector: 'app-tabbed-pane',
  templateUrl: './tabbed-pane.component.html',
  styleUrls: ['./tabbed-pane.component.scss']
})
export class TabbedPaneComponent implements OnInit, AfterContentInit {

  @ContentChildren(TabComponent)
  tabQueryList: QueryList<TabComponent> | undefined;

  activeTab: TabComponent | undefined;
  currentPage = 0;

  get tabs(): TabComponent[] {
    return this.tabQueryList?.toArray() ?? [];
  }

  ngAfterContentInit(): void {
    if (this.tabs.length > 0) {
      this.activate(this.tabs[0]);
    }
  }
}

[...]
}

```

Zum direkten Zugriff auf den Content und die darin positionierten Registerblätter definiert die Komponente eine Eigenschaft `tabQueryList` vom Typ `QueryList`. Der Dekorator `ContentChildren` gibt an, dass die `QueryList` mit den einzelnen `TabComponents` aus dem Content zu bestücken ist. Deswegen bekommt sie den Typ `TabComponent` als Filter übergeben.

Das ursprüngliche Array `tabs` wurde gegen einen Getter getauscht, der den Inhalt der `QueryList` als herkömmliches Array zurückliefert. Die Methode `register` und deren Aufruf in der `TabComponent` kann somit entfallen:

```

// src/app/shared/controls/tab/tab.component.ts

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

@Component([...])
export class TabComponent implements OnInit {

  @Input() title = '';
  visible = true;

  ngOnInit(): void {
  }
}

```

Neben dem Dekorator ContentChildren existiert mit ContentChild auch eine Alternative für Fälle, in denen man genau ein einziges Element aus dem Content abfragen möchte:

```
@ContentChild(TabComponent)
lonelyTabComponent: TabComponent | undefined;
```

Beide Dekoratoren können auch eine Template-Variable als Filter nutzen. Das nachfolgende Beispiel ermittelt damit sämtliche Elemente im Content, die die Template-Variable myVar aufweisen:

```
@ContentChildren('myVar')
tabQueryList: QueryList<TabComponent> | undefined;
```

Der Content könnte sich wie folgt gestalten:

```
<app-tab title="A" #myVar></app-tab>
<app-tab title="B" #myVar></app-tab>
```

Analog dazu lässt sich mit ContentChild ein einziges Element aus dem Content mit der angegebenen Template-Variablen abfragen:

```
@ContentChild('myVar')
lonelyTabComponent: TabComponent | undefined;
```

## Interaktion mit der View

Ähnlich, wie eine Komponente mit den Dekoratoren ContentChild und Content Children ihren Content abfragen kann, kann sie ViewChild und ViewChildren zum Abfragen ihrer View einsetzen.

Um das zu demonstrieren, führen wir hier eine TabNavigatorComponent ein. Sie soll die Möglichkeit bieten, zwischen den Registerblättern einer TabbedPane hin und her zu blättern (siehe Abbildung 18-4).



Abbildung 18-4: Die TabNavigatorComponent erlaubt das Blättern zwischen den Registerblättern.

Zum Generieren der Komponente verwenden wir wieder die CLI:

```
ng g c shared/controls/tab-navigator --export
```

Der Aufrufer übergibt in der Implementierung der Komponente die aktuelle Seite an page – z. B. 1, wenn das erste Registerblatt aktiv ist – sowie die Anzahl der Registerblätter an pageCount (siehe Beispiel 18-11).

*Beispiel 18-11: TabNavigatorComponent zum Blättern zwischen Registerblättern*

```
// src/app/shared/controls/tab-navigator/tab-navigator.component.ts

import { Component, EventEmitter, Input, OnInit, Output } from '@angular/core';

@Component({
  selector: 'app-tab-navigator',
  templateUrl: './tab-navigator.component.html',
  styleUrls: ['./tab-navigator.component.scss']
})
export class TabNavigatorComponent implements OnInit {

  @Input() page = 0;
  @Input() pageCount = 0;
  @Output() pageChange = new EventEmitter<number>();

  constructor() { }

  ngOnInit(): void {
  }

  prev(): void {
    if (this.page <= 1) {
      return;
    }
    this.page--;
    this.pageChange.emit(this.page);
  }

  next(): void {
    if (this.page >= this.pageCount) {
      return;
    }
    this.page++;
    this.pageChange.emit(this.page);
  }
}
```

Die Methoden prev und next implementieren das Blättern, und pageChange informiert über Seitenwechsel, die durch das Blättern auftreten.

Das Template präsentiert lediglich die aktuelle Seite und bietet Schaltflächen, die an prev und next gebunden sind:

```
<!-- src/app/shared/controls/tab-navigator/tab-navigator.component.html -->

<div class="tab-navigator">
  <button class="prev" (click)="prev()">&lt;&lt;</button>
  # {{page}}
  <button class="next" (click)="next()">&gt;&gt;</button>
</div>
```

Die Styles dieser Komponente finden sich in Beispiel 18-12.

*Beispiel 18-12: Styles für TabNavigatorComponent*

```
/* src/app/shared/controls/tab-navigator/tab-navigator.component.scss */

.tab-navigator {
  border: 2px solid black;
  width:150px;
}

.tab-navigator button {
  border:none;
  background-color: inherit;
}

.tab-navigator .next {
  float: right;
}
```

Die neue TabNavigatorComponent kann nun am Ende des Templates der TabbedPane Component aufgerufen werden:

```
<!-- src/app/shared/controls/tabbled-pane/tabbled-pane.component.html -->

<div class="tabbed-pane">
  [...]
  <app-tab-navigator
    [page]="this.currentPage"
    [pageCount]="this.tabs.length"
    (pageChange)="pageChange($event)">
  </app-tab-navigator>
</div>
```

Um das pageChange-Event zu behandeln, erhält die TabbedPaneComponent einen gleichnamigen Event-Handler. Außerdem verwaltet sie nun die aktuelle Seitennummer in der Eigenschaft currentPage (siehe Beispiel 18-13).

*Beispiel 18-13: Event-Handler für die TabbedPaneComponent*

```
// src/app/shared/controls/tabbled-pane/tabbled-pane.component.ts

import { AfterContentInit, Component, ContentChildren, OnInit, QueryList }
  from '@angular/core';
import { TabComponent } from '../tab/tab.component';

@Component({
  selector: 'app-tabbled-pane',
  templateUrl: './tabbed-pane.component.html',
  styleUrls: ['./tabbed-pane.component.scss']
})
export class TabbedPaneComponent implements OnInit, AfterContentInit {
```

```

@ContentChildren(TabComponent)
tabQueryList: QueryList<TabComponent> | undefined;

activeTab: TabComponent | undefined;

// Hinzufügen:
currentPage = 0;

get tabs(): TabComponent[] {
  return this.tabQueryList?.toArray() ?? [];
}

constructor() {

ngAfterContentInit(): void {
  if (this.tabs.length > 0) {
    this.activate(this.tabs[0]);
  }
}

ngOnInit(): void {

activate(active: TabComponent): void {
  for (const tab of this.tabs) {
    tab.visible = (tab === active);
  }
  this.activeTab = active;

  // Hinzufügen:
  this.currentPage = this.tabs.indexOf(active) + 1;
}

// Hinzufügen:
pageChange(page: number): void {
  this.activate(this.tabs[page - 1]);
}

}
}

```

Die Kommunikation mit der TabNavigatorComponent erfolgt hier über Datenbindung. Alternativ dazu könnte die TabbedPaneComponent programmatisch auf ihre View zugreifen, die TabNavigatorComponent referenzieren und direkt ihre Eigenschaften verändern. Um das zu demonstrieren, entfernen wir ein paar der Datenbindungen:

```

<!-- src/app/shared/controls/tabbed-pane/tabbed-pane.component.html -->

<div class="tabbed-pane">
  [...]

    <app-tab-navigator [page]="currentPage" #navigator>
      </app-tab-navigator>
    </div>

```

Um eine Referenz auf ihre TabNavigatorComponent zu erhalten, nutzt die TabbedPane Component nun den Dekorator ViewChild (siehe Beispiel 18-14).

*Beispiel 18-14: View mit ViewChild abfragen*

```
// src/app/shared/controls/tabbled-pane/tabbled-pane.component.ts

// ViewChild importieren:
import {
    AfterContentInit,
    AfterViewInit,
    Component,
    ContentChildren,
    OnInit,
    QueryList,
    ViewChild
}
from '@angular/core';

// Hinzufügen:
import { TabNavigatorComponent } from '../tab-navigator/tab-navigator.component';

import { TabComponent } from '../tab/tab.component';

@Component([...])
export class TabbedPaneComponent implements OnInit, AfterContentInit, AfterViewInit {

    @ContentChildren(TabComponent)
    tabQueryList: QueryList<TabComponent> | undefined;

    // Einfügen:
    @ViewChild('navigator')
    navigator: TabNavigatorComponent | undefined;

    activeTab: TabComponent | undefined;
    currentPage = 0;

    get tabs(): TabComponent[] {
        return this.tabQueryList?.toArray() ?? [];
    }

    constructor() {}

    // Direkt mit Navigator interagieren:
    ngAfterViewInit(): void {
        if (this.navigator) {
            this.navigator.pageCount = this.tabs.length;
            // Diese Anweisung würde einen Zyklus verursachen:
            // this.navigator.page = 1;
            this.navigator.pageChange.subscribe((page: number) => {
                this.pageChange(page);
            });
        }
    }

    ngAfterContentInit(): void {
```

```

        if (this.tabs.length > 0) {
            this.activate(this.tabs[0]);
        }
    }

ngOnInit(): void {
}

[...]
}

```

Dieses Beispiel übergibt an den Dekorator `ViewChild` den Namen der Template-Variablen `navigator` als Filterkriterium. Mit der so erhaltenen `TabNavigatorComponent` kommuniziert sie innerhalb des Life-Cycle-Hooks `AfterViewInit` (nicht zu verwechseln mit `AfterContentInit`), den Angular nach dem Initialisieren der View anstößt.

Neben `ViewChild` existiert analog zum oben gezeigten `ContentChildren` auch ein Dekorator `ViewChildren`. Beide akzeptieren als Filter sowohl Typen als auch Template-Variablen.

Bitte beachten Sie, dass das direkte Setzen der Eigenschaft `page` in `ngAfterViewInit` zu einem Zyklus führen würde. Die View wurde an dieser Stelle ja schon gerendert, und ein nachträgliches Verändern würde ein erneutes Rendering mit sich bringen. Wie in Kapitel 4 besprochen, erlaubt Angular so etwas aus Gründen der Performance und Nachvollziehbarkeit nicht und mahnt den Versuch mit einer Exception ab.

## Den Sinn von `ViewChild` und `ViewChildren` hinterfragen

Wie im letzten Abschnitt klar wurde, lässt sich die herkömmliche Datenbindung durch eine direkte Interaktion mit der View ersetzen. `ViewChild` und `ViewChildren` erlauben es dazu, einzelne Elemente der View abzufragen.

In den meisten Fällen ist das jedoch keine gute Idee! Mit dieser Vorgehensweise ersetzt man eine Kernidee von Angular, nämlich die deklarative Datenbindung, durch eigenen Code. Das macht den Code schwieriger nachvollziehbar und somit auch schwieriger wartbar. Außerdem kann dieses Vorgehen, wie gezeigt, zu Zyklen führen.

Wir hinterfragen in unseren Codereviews die Nutzung sämtlicher `ViewChild`- und `ViewChildren`-Aufrufe. Wir sehen deren Einsatz nur dann gerechtfertigt, wenn ein Drittanbietersteuerelement für bestimmte Eigenschaften keine Datenbindung erlaubt und wenn ein Parent eine Methode einer Child-Komponente aufrufen muss.

## Life-Cycle-Hooks für View und Content

In den vorausgegangenen Abschnitten haben wir mit `ngAfterViewInit` und `ngAfterContentInit` schon zwei Life-Cycle-Hooks, die sich auf die View und den Content beziehen, verwendet. Angular bietet allerdings noch zwei weitere. Tabelle 18-1 fasst alle vier zusammen.

Tabelle 18-1: Life-Cycle-Hooks für View und Content

Interface	Methode	Beschreibung
AfterContentInit	ngAfterContentInit	Wird nach dem Initialisieren des Contents und somit nach der initialen Content Projection aufgerufen.
AfterContentChecked	ngAfterContentChecked	Wird aufgerufen, nachdem Angular im Rahmen der Change Detection den Content auf Änderungen geprüft hat.
AfterViewInit	ngAfterViewInit	Wird aufgerufen, nachdem Angular die View initialisiert hat. Im Rahmen dessen findet auch die initiale Datenbindung statt.
AfterViewChecked	ngAfterViewChecked	Wird aufgerufen, nachdem Angular im Rahmen der Change Detection den View Content auf Änderungen geprüft hat.

Interessant zu wissen ist, dass Angular all diese Life-Cycle-Hooks nach OnInit sowie auch nach OnChanges anstößt! Die nachfolgende Auflistung zeigt die Aufrufreihenfolge:

1. ngOnChanges
2. ngOnInit
3. ngDoCheck
4. ngAfterContentInit
5. ngAfterContentChecked
6. ngAfterViewInit
7. ngAfterViewChecked
8. ngOnDestroy

Abgesehen von ngOnInit und ngOnDestroy führt Angular diese Hooks im Rahmen jeder Change Detection aus. Der Hook ngOnInit wird nur einmal nach dem Initialisieren der Komponente angestoßen und ngOnDestroy lediglich, wenn Angular die Komponente zerstört.

Das bedeutet, dass sich die Komponente erst nach ihrer Initialisierung um ihren Content und ihre View kümmert.

## Statische Child-Komponenten

Bis jetzt haben wir die Dekoratoren ViewChild, ViewChildren, ContentChild und ContentChildren lediglich mit einem Argument, das als Filter diente, aufgerufen. Allerdings bieten diese Dekoratoren noch ein weiteres Argument – ein Parameterobjekt zum Anpassen des Verhaltens.

Die Eigenschaft static gibt an, ob die gewünschten Elemente vor oder nach der Datenbindung ermittelt werden sollen:

```
@ViewChild('navigator', { static: true })
navigator: TabNavigatorComponent | undefined;
```

Existieren diese Elemente von Anfang an, z.B. weil die Anwendung sie nicht erst mit einem \*ngIf oder \*ngFor erzeugen muss, können Sie diese Eigenschaft auf true setzen. In dem Fall bekommen Sie bereits beim Aufruf von ngOnInit Zugriff darauf.

Ansonsten stehen Content-Children erst beim Aufruf von AfterContentChecked bzw. AfterContentInit zur Verfügung. AfterContentChecked verwenden Sie, wenn sich die Menge der referenzierten Elemente im Rahmen der Change Detection ändern kann.

Analog dazu nutzen Sie für View-Children AfterViewChecked bzw. AfterViewInit.

### Zugriff auf das DOM mit der ElementRef

Neben static bietet das Parameterobjekt von ViewChild, ViewChildren, Content Child und ContentChildren mit der Eigenschaft read die Möglichkeit, alternative Repräsentationen der Child-Komponenten zu laden. Damit lässt sich zum Beispiel die ElementRef ermitteln. Dabei handelt es sich um ein Wrapper-Objekt für den nativen DOM-Knoten, der die Child-Komponente repräsentiert:

```
@ViewChild('navigator', { read: ElementRef, static: true })
navigatorElementRef: ElementRef | undefined;
```

Die Eigenschaft nativeElement referenziert den DOM-Knoten:

```
if (this.navigatorElementRef) {
  this.navigatorElementRef.nativeElement.style.color = 'red';
}
```

Um das ElementRef für die aktuelle Komponente zu erhalten, fordern Sie es per Dependency Injection an:

```
constructor(private elm: ElementRef) {
  elm.nativeElement.style.color = 'red';
}
```

Generell sollten Sie native DOM-Zugriffe so weit wie möglich vermeiden, zumal Sie damit die Möglichkeiten von Angular umschiffen und aushebeln.

### Zugriff auf die ViewContainerRef

Neben der ElementRef bietet Angular mit der ViewContainerRef eine weitere Repräsentation für Komponenten. Sie spiegelt den Bereich wider, den eine Komponente einnimmt, und erlaubt es, diesen Bereich zu löschen oder um zusätzliche Elemente und Komponenten zu erweitern.

Das folgende Beispiel nutzt diese Möglichkeit, um den ViewContainer um eine BasketComponent zu erweitern:

```
constructor(private container: ViewContainerRef, cfr: ComponentFactoryResolver) {
  container.createComponent(cfr.resolveComponentFactory(BasketComponent));
}
```

Aus Angular-internen Gründen müssen Sie hier eine sogenannte ComponentFactory für die BasketComponent angeben. Diese erhalten Sie über einen ComponentFactory Resolver, der sich ebenfalls injizieren lässt.

Um die ViewContainerRef für View- bzw. Content-Children zu erhalten, nutzen Sie die Option read von ViewChild, ViewChildren, ContentChild und ContentChildren:

```
@ViewChild('navigator', { read: ViewContainerRef, static: true })
navigatorViewContainerRef: ViewContainerRef | undefined;
```

Mehr Informationen dazu präsentieren wir im Zusammenhang mit Direktiven in Kapitel 18.

## Kommunikation über Template-Variablen

Eine Template-Variable, die ein Template für eine Komponente einrichtet, bietet auch eine Referenz auf die Komponenteninstanz. Somit gewährt sie Zugriff auf die Eigenschaften und Methode der Komponente. Beispiel 18-15 nutzt zum Beispiel die Template-Variable navigator.

*Beispiel 18-15: Kommunikation über Template-Variablen*

```
<!-- src/app/shared/controls/tabbed-pane/tabbed-pane.component.html -->
<div class="tabbed-pane">
  [...]
  <app-tab-navigator [page]="currentPage" #navigator>
  </app-tab-navigator>
  <div>
    <button (click)="navigator.prev()">Next</button>
    {{navigator.page}}
    <button (click)="navigator.next()">Prev</button>
  </div>
</div>
```

Über diese Template-Variable ruft das Beispiel die Methoden prev und next der TabNavigatorComponent auf und liest deren Eigenschaft page.



Handelt es sich bei einem Element nicht um eine Angular-Komponente, verweist eine dafür eingerichtete Template-Variable auf den jeweiligen DOM-Knoten:

```

<button (click)="myImage.src = '...'"></button>
```

## Kommunikation über Services

Eine weitere Möglichkeit zur Kommunikation zwischen Komponenten ist der Einsatz von Services. Wie in Kapitel 5 beschrieben, kann eine Anwendung Services für

eine Komponente einrichten. Die Serviceinstanz kann sowohl die Komponente als auch alle ihre Child-Komponenten (und deren Child-Komponenten etc.) nutzen. Somit können alle Komponenten eines Teils des Komponentenbaums der Anwendung über den Service kommunizieren.

Um diese Idee zu demonstrieren, führen wir hier einen Service zur Kommunikation zwischen TabbedPaneComponent und TabNavigatorComponent ein:

```
ng g s shared/controls/tabbed-pane/tabbed-pane
```

Die Implementierung des Service stellt lediglich zwei BehaviorSubjects bereit: pageCount repräsentiert die Anzahl an Seiten insgesamt und currentPage die aktuelle Seite (siehe Beispiel 18-16).

*Beispiel 18-16: TabbedPaneService*

```
// src/app/shared/controls/tabbed-pane/tabbed-pane.service.ts
```

```
import { Injectable } from '@angular/core';
import { BehaviorSubject, Subject } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class TabbedPaneService {

  readonly pageCount = new BehaviorSubject<number>(0);
  readonly currentPage = new BehaviorSubject<number>(1);

  constructor() { }

}
```

Diesen Service registrieren wir bei der TabbedPaneComponent. Dazu kommt die Eigenschaft providers im Component-Dekorator zum Einsatz (siehe Beispiel 18-17).

*Beispiel 18-17: Die TabbedPaneComponent lässt sich den TabbedPaneService injizieren.*

```
// src/app/shared/controls/tabbed-pane/tabbed-pane.component.ts

[...]

// Import hinzufügen:
import { TabbedPaneService } from './tabbed-pane.service';

@Component({
  [...]

  // Provider hinzufügen:
  providers: [TabbedPaneService]
})

export class TabbedPaneComponent implements OnInit, AfterContentInit, AfterViewInit {

  [...]

  constructor(private service: TabbedPaneService) {
```

```
}

[...]

}
```

Außerdem lässt sich die TabbedPaneComponent den TabbedPaneService injizieren. Sie nutzt ihn, um sich über Änderungen an der Eigenschaft currentPage zu informieren. Ändert sie selbst die aktuelle Seite, informiert sie die TabNavigatorComponent auch über den Service (siehe Beispiel 18-18).

*Beispiel 18-18: Die TabbedPaneComponent teilt Eckdaten mit dem TabbedPaneService.*

```
// src/app/shared/controls/tabbed-pane/tabbed-pane.component.ts
[...]

// Import hinzufügen:
import { TabbedPaneService } from './tabbed-pane.service';

[...]

// Diese Methode aktualisieren:
ngAfterViewInit(): void {
    this.service.pageCount.next(this.tabs.length);
    this.service.currentPage.subscribe((page: number) => {
        // Zyklus vermeiden!
        if (page === this.currentPage) {
            return;
        }
        this.pageChange(page);
    });
}
[...]

activate(active: TabComponent): void {
    for (const tab of this.tabs) {
        tab.visible = (tab === active);
    }
    this.activeTab = active;
    // Aktualisieren:
    this.currentPage = this.tabs.indexOf(active) + 1;
    this.service.currentPage.next(this.currentPage);
}
```

Außerdem informiert die TabbedPaneComponent den Service über die gesamte Seitenanzahl.

Auch die TabNavigatorComponent lässt sich den TabbedPaneService injizieren und nutzt ihn, um sich über die aktuelle Seite und die Seitenanzahl zu informieren. Ändert sie die aktuelle Seite, informiert sie den Service und somit indirekt die TabbedPaneComponent (siehe Beispiel 18-19).

*Beispiel 18-19: Die TabNavigatorComponent erhält nun Informationen über den TabbedPaneService.*

```
// src/app/shared/controls/tab-navigator/tab-navigator.component.ts

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

//Hinzufügen:
import { TabbedPaneService } from '../tabbed-pane/tabbed-pane.service';

@Component({
  selector: 'app-tab-navigator',
  templateUrl: './tab-navigator.component.html',
  styleUrls: ['./tab-navigator.component.scss']
})
export class TabNavigatorComponent implements OnInit {

  // Keine Inputs mehr.
  page = 0;
  pageCount = 0;

  // Output wurde entfernt!

  // Hinzufügen: Service injizieren lassen.
  constructor(private service: TabbedPaneService) { }

  ngOnInit(): void {
    // Hinzufügen: Von Service benachrichtigen lassen.
    this.service.pageCount.subscribe(pageCount => {
      this.pageCount = pageCount;
    });
    this.service.currentPage.subscribe(page => {
      this.page = page;
    });
  }

  prev(): void {
    if (this.page <= 1) {
      return;
    }
    this.page--;
  }

  // Hinzufügen: Service benachrichtigen.
  this.service.currentPage.next(this.page);
}

next(): void {
  if (this.page >= this.pageCount) {
    return;
  }
  this.page++;
}

  // Hinzufügen: Service benachrichtigen.
  this.service.currentPage.next(this.page);
}

}
```

Der Vorteil von dieser Art der Kommunikation ist die lose Kopplung. Die Tabbed PaneComponent muss gar nicht wissen, welche anderen Komponenten sich für ihre Informationen interessiert. Sie versendet sie einfach über das Observable. Somit lässt sich diese Art der Kommunikation gut erweitern.

Außerdem erleichtert die Nutzung eines Service die Kommunikation über verschiedene Hierarchieebenen des Komponentenbaums hinweg.

Der Nachteil dieses Ansatzes liegt darin, dass die Kommunikation implizit erfolgt. Man merkt auf den ersten Blick nicht, wer mit wem kommuniziert. Außerdem dürfte es den Konsumenten solch einer Komponente verwirren, wenn er oder sie über einen Service mit der Komponente kommunizieren muss. Datenbindungen sind an dieser Stelle offensichtlicher.

## Attributdirektiven

Bei den in Angular omnipräsen Komponenten handelt es sich genau genommen lediglich um eine konkrete Ausprägung eines allgemeineren Konzepts, das sich *Direktive* nennt. Eine solche Direktive ergänzt bestimmte Elemente der Seite um zusätzliches Verhalten. Ein Beispiel dafür ist `ngModel`, das Datenbindungsverhalten hinzufügt, oder `ngStyle`, das Verhalten in Hinblick auf Formatierungen definiert. Die zu adressierenden Elemente definiert eine Direktive über ihren Selektor. Eine Komponente macht dasselbe, fügt jedoch auch eine View den adressierten Elementen hinzu. Diese Elemente definiert die Komponente bekanntermaßen über ihr Template. Somit könnte man sagen, dass eine Komponente lediglich eine Direktive mit einem Template ist.

In diesem Abschnitt betrachten wir die einfachste Form von Direktiven, die sogenannten Attributdirektiven. Der Name spiegelt die Tatsache wider, dass ihre Selektoren in der Regel Elemente mit bestimmten Attributen adressieren. Allerdings ist dies keine Vorschrift, sondern lediglich eine gelebte Konvention.

Um dieses Konzept zu demonstrieren, stellen wir eine Alternative zum klassischen `click`-Ereignis bereit, die für kritische Aktionen gedacht ist. Sie gibt zunächst lediglich eine Warnmeldung aus und stößt den hinterlegten Event-Handler nur dann an, wenn diese Warnung bestätigt wurde.

## Direktiven definieren

Eine Direktive definieren Sie ähnlich wie eine Komponente: Es handelt sich dabei um eine Klasse, die Bindings aufweisen kann. Metadaten sind über den Dekorator Directive bereitzustellen. Dieser hat fast alle Eigenschaften, die Sie auch von Component kennen – lediglich Template-bezogene Eigenschaften fehlen, zumal Direktiven eben keine Templates haben. Zu diesen Eigenschaften, die man vergeblich suchen wird, zählen template bzw. templateUrl, styles bzw. styleUrls und view Providers.

Die hier betrachtete Direktive lässt sich mit der CLI erstellen:

```
ng g directive shared/controls/click-with-warning --export
```

Stellen Sie auch sicher, dass Ihr SharedModule die neue Direktive sowohl deklariert als auch exportiert (siehe Beispiel 18-20).

*Beispiel 18-20: Die ClickWithWarningDirective wird vom SharedModule deklariert und exportiert.*

```
// src/app/shared/shared.module.ts
[...]

// Import hinzufügen:
import { ClickWithWarningDirective } from './controls/click-with-warning.directive';

@NgModule({
  imports: [
    [...]
  ],
  declarations: [
    [...],
    ClickWithWarningDirective
  ],
  exports: [
    [...],
    ClickWithWarningDirective
  ]
})
export class SharedModule { }
```

Die hier betrachtete Direktive erhält einen Selektor, der sämtliche Elemente mit dem Attribut flightClickWithWarning adressiert (siehe Beispiel 18-21).

*Beispiel 18-21: Beispiel für eine Attributdirektive*

```
// src/app/shared/controls/click-with-warning.directive.ts

// Imports aktualisieren:
import { Directive, ElementRef, OnInit } from '@angular/core';

@Directive({
  selector: '[appClickWithWarning]'
})
export class ClickWithWarningDirective implements OnInit {

  constructor(private elementRef: ElementRef) {
  }

  ngOnInit(): void {
    this.elementRef.nativeElement.setAttribute('class', 'btn btn-danger');
  }
}
```

Bitte achten Sie beim Selektor auf die eckigen Klammern. Diese legen fest, dass es sich bei `appClickWithWarning` um ein Attribut handelt. Die Direktive bezieht sich somit auf alle Elemente mit einem `appClickWithWarning`-Attribut. Solch ein Element wird auch als Host bezeichnet.

Da sich Angular bei den Selektoren an den Möglichkeiten von CSS orientiert, können Sie hier auch komplexere Konstrukte zum Identifizieren der gewünschten Host-Elemente verwenden. Beispielsweise würde der Selektor

```
button[appClickWithWarning]
```

lediglich `button`-Elemente mit einem `appClickWithWarning`-Attribut adressieren, und im Fall von

```
div.container button[appClickWithWarning]
```

müsste sich dieses `button`-Element auch innerhalb eines `div` mit der Klasse `container` befinden.

Die hier gezeigte Direktive lässt sich die aktuelle  `ElementRef` injizieren. Dabei handelt es sich um ein Objekt, das den Host referenziert.

Die  `ElementRef` gibt uns mit ihrer Eigenschaft `nativeElement` Zugriff auf den zu-grunde liegenden DOM-Knoten. Das Beispiel nutzt diesen, um die Klassen `btn` und `btn-danger` dem Button zuzuweisen.

Generell sollten Sie direkte DOM-Zugriffe so weit wie möglich meiden, weil diese das Framework umschiffen. Das führt zu schlecht wartbarem Code. Außerdem funktionieren DOM-Zugriffe nicht in jeder Umgebung. Wollen Sie beispielsweise die erste Ansicht Ihrer Anwendung serverseitig vorrendern, kann es hier zu Problemen kommen. Serverseitig simuliert Angular das DOM zwar, aber diese Vorgehensweise hat natürlich ihre Grenzen.

Um die Direktive auszuprobieren, ist lediglich eine Schaltfläche mit dem Attribut `appClickWithWarning` zu versehen.

```
<!-- src/app/customer/booking-history/booking-history.component.html -->
```

```
[...]
<button appClickWithWarning>Delete</button>
[...]
```

## Mit der Umwelt kommunizieren

Analog zu Komponenten können Direktiven über Datenbindungen mit ihrem Aufrufer kommunizieren. Dazu nutzen auch sie Inputs und Outputs.

Daneben können Direktiven direkten Zugriff auf die Eigenschaften und Events ihres Hosts erlangen. Das bedeutet, die Direktive kann Event-Handler für Events des Hosts einrichten und seine Eigenschaften verändern. Für Ersteres bietet Angular sogenannte `HostListener`, für Letzteres `HostBindings` (siehe Beispiel 18-22).

*Beispiel 18-22: Direktive mit HostListener und HostBinding*

```
// src/app/shared/controls/click-with-warning.directive.ts

// Imports aktualisieren:
import {
  Directive,
  ElementRef,
  EventEmitter,
  HostBinding,
  HostListener,
  Input,
  OnInit,
  Output
}
from '@angular/core';

@Directive({
  selector: '[appClickWithWarning]'
})
export class ClickWithWarningDirective implements OnInit {

  // Input und Output ergänzen:
  @Input() warning = 'Are you sure?';
  @Output() appClickWithWarning = new EventEmitter();

  // HostBinding ergänzen:
  @HostBinding('class') classBinding: string | undefined;

  constructor(private elementRef: ElementRef) {
  }

  // HostListener ergänzen:
  @HostListener('click', ['$event'])
  handleClick($event: MouseEvent): void {
    if ($event.shiftKey || confirm(this.warning)) {
      this.appClickWithWarning.emit();
    }
  }
}

ngOnInit(): void {
  // Klassen über HostBinding zuweisen:
  this.classBinding = 'btn btn-danger';
}

}
```

Der Name `appClickWithWarning` wird in dieser aktualisierten Version unserer Direktive nicht nur für den Selektor herangezogen, sondern auch für das Event, das nach einer eventuellen Bestätigung aufzurufen ist. Eine solche Vorgehensweise ist üblich, zumal sie es erlaubt, im selben Atemzug sowohl die Direktive auf eine Komponente anzuwenden als auch eine erste Bindung festzulegen:

```
<!-- src/app/customer/booking-history/booking-history.component.html -->
```

```
[...]
```

```
<button (appClickWithWarning)="delete()">Delete</button>
[...]
```

Der hier betrachtete HostListener bringt bei jedem click-Ereignis des Hosts die Methode handleClick zur Ausführung. Diese gibt eine Warnung aus und stößt – sofern sie bestätigt wird – das Ereignis flightClickWithWarning an.

Neben dem Namen des gewünschten Ereignisses nimmt der HostListener-Dekorator auch ein Array mit Objekten entgegen, die an den dekorierten Handler weiterzureichen sind. Im betrachteten Fall wurde lediglich das aktuelle Eventobjekt angefordert. Alternativ dazu könnte der HostListener auch andere verfügbare Objekte oder eine Projektion des Eventobjekts referenzieren. Auf diese Weise könnten Sie sich zum Beispiel direkt den Wert für die Eigenschaft shiftKey übergeben lassen:

```
@HostListener('click', ['$event.shiftKey'])
handleClick(shiftKey: boolean): void {
  if (shiftKey || confirm(this.warning)) {
    this.appClickWithWarning.emit();
  }
}
```

Analog zum HostListener gibt uns das HostBinding direkten Zugriff auf Eigenschaften des Hosts. Sie können darüber die Eigenschaften auslesen, aber auch verändern. Auf diese Weise erhält die hier betrachtete Direktive Zugriff auf die zugewiesenen Klassen. In ihrer Methode ngOnInit setzt sie diese auf btn btn-danger. Damit lässt sich der oben beschriebene direkte DOM-Zugriff vermeiden.

## Direktiven und Template-Variablen

Auch Direktiven lassen sich über Template-Variablen ansprechen. Hier ergibt sich jedoch eine kleine Herausforderung: Hinter einem Element im Template können neben einer Komponente gleich mehrere Direktiven stehen. Wir benötigen also einen Weg, Angular wissen zu lassen, ob wir die Komponente oder eine der Direktiven meinen.

Um das zu ermöglichen, müssen Sie Ihrer Direktive mit exportAs einen möglichst eindeutigen Wert zuweisen:

```
@Directive({
  selector: '[appClickWithWarning]',
  exportAs: 'clickWithWarning'
})
export class ClickWithWarningDirective implements OnInit {
  [...]
}
```

Hier lautet er clickWithWarning. Auf diesen Wert können Sie sich beim Deklarieren der Template-Variablen beziehen:

```
<button (appClickWithWarning)="delete()" #cww="clickWithWarning">Delete</button>
<button (click)="cww.handleClick(true)">Don't ask questions!</button>
```

In diesem Beispiel erhält die Template-Variable `cww` Zugriff auf die `appClickWithWarning`-Direktive der Schaltfläche.

Sie können `exportAs` übrigens auch mit Komponenten verwenden. Das ist aber in der Regel nicht nötig, da Angular die Komponenteninstanz standardmäßig einer Template-Variablen zuweist, wenn Sie keinen expliziten Wert anführen. Weiter oben haben wir diese Möglichkeit bereits zum Zugriff auf die `TabNavigatorComponent` verwendet:

```
<app-tab-navigator [page]="currentPage" #navigator>
</app-tab-navigator>
```



Eventuell erinnern Sie sich daran, wie wir in Kapitel 9 Zugriff auf den Objektgraphen, der ein Formular beschreibt, bekommen haben:

```
<form #form="ngForm">
  {{form.valid}}
  [...]
</form>
```

Hier nutzt Angular intern die gleichen Mechanismen: Angular platziert eine Direktive auf jedem `form`-Element und exportiert sie unter dem Namen `ngForm`.

## Strukturelle Direktiven

Strukturelle Direktiven verändern den Inhalt der Elemente, auf die sie angewandt wurden – oder anders ausgedrückt: ihre Struktur. Dazu nutzen sie in der Regel Templates, die sie bei Bedarf beliebig häufig rendern. Bekannte Vertreter, die direkt mit Angular ausgeliefert werden, sind `ngIf` und `ngFor`.

In diesem Abschnitt erklären wir zunächst die zugrunde liegende Funktionsweise anhand von zwei einfachen Beispielen, mit denen wir einen Teil der Funktionalität von `ngIf` und `ngFor` nachstellen. Die erklärten Konzepte nutzen wir anschließend für ein paar Anwendungsfälle, die sich bei der Schaffung von Komponentenbibliotheken häufig ergeben.

### Templates und Container

Der einzige wirkliche Unterschied zu Attributdirektiven ist, dass strukturelle Direktiven ihr eigenes Template haben. Dabei handelt es sich um jenes Element, auf das sie angewendet werden. Im Fall von

```
<div *ngIf="flights.length > 0">Flüge!</div>
```

handelt es sich beim gesamten `div` um das von `ngIf` verwendete Template. Die Direktive kann nun entscheiden, ob, wann und wie häufig es den Inhalt rendert.

Um zu zeigen, wie strukturelle Direktiven umgesetzt werden, stellen wir vereinfacht das Verhalten von `ngIf` mit einer `unless`-Direktive nach. Im Gegensatz zu `ngIf`

rendert unless ihren Inhalt jedoch nur, wenn das übergebene Kriterium auf false ausgewertet werden kann. Insofern verfolgt sie eine Semantik im Sinne von »if not«.

Die Direktive lässt sich mit der CLI erzeugen:

```
ng generate directive shared/common/unless --export
```

Ihre Implementierung findet sich in Beispiel 18-23.

*Beispiel 18-23: Eine einfache strukturelle Direktive*

```
// src/app/shared/common/unless.directive.ts

import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';

@Directive({
  selector: '[appUnless]'
})
export class UnlessDirective {

  @Input() set appUnless(condition: boolean) {
    if (!condition) {
      this.viewContainer.createEmbeddedView(this.templateRef);
    } else {
      this.viewContainer.clear();
    }
  }
  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef
  ) {
  }
}
```

Per Dependency Injection nimmt unsere unless-Direktive eine TemplateRef sowie eine ViewContainerRef entgegen. Erstere referenziert das Element, auf das die strukturelle Direktive angewandt wird. Dieses wird, wie von ngIf oder ngFor bekannt, als Template genutzt. Das bedeutet, dass es nicht sofort gerendert wird, sondern erst, wenn die Direktive es für richtig befindet. Die ViewContainerRef referenziert hingegen das betroffene Element und erlaubt das Einfügen von Templates.

Der Selektor adressiert sämtliche Elemente mit dem Attribut appUnless. Es existiert auch ein Input-Binding mit demselben Namen. Somit kann durch Angabe dieses Namens die Direktive auf ein Element angewandt sowie eine Datenbindung durchgeführt werden. Diese Datenbindung nimmt das auszuwertende Kriterium entgegen. Ist es false, rendert der Setter das Template im ViewContainer. Ansonsten wird der ViewContainer mit clear geleert, was zur Folge hat, dass Angular keinen Inhalt anzeigt.

Um die strukturelle Direktive zu nutzen, ist sie streng genommen auf ein Template anzuwenden. Wie wir schon angemerkt haben, rendert Angular ein Template nicht sofort, sondern überlässt diese Aufgabe den einzelnen Direktiven:

```
<!-- src/app/customer/booking-history/booking-history.component.html -->

[...]
<ng-template [appUnless]="flights.length == 0">
  <div>{{flights.length}} Flüge!</div>
</ng-template>
[...]
```

Der Einsatz von Templates bringt leider auch wortreiche Codestrecken mit sich. Aus diesem Grund bietet Angular hierfür ein wenig syntaktischen Zucker:

```
<div template="appUnless: flights.length == 0">
  <div>{{flights.length}} Flüge!</div>
</div>
```

Aber auch diese Schreibweise ist noch länger, als sie sein müsste, und deswegen erlaubt Angular hierfür die von `ngIf` und `ngFor` bekannte Schreibweise, die mit einem Stern eingeleitet wird:

```
<div *appUnless="flights.length == 0">
  <div>{{flights.length}} Flüge!</div>
</div>
```

## Microsyntax

Die im letzten Abschnitt vorgestellte Direktive hat ihr Template entweder einmal oder keinmal gerendert. Eine strukturelle Direktive kann ein Template jedoch auch mehrfach rendern. Darüber hinaus kann sie über eine sogenannte Microsyntax zusätzliche Bindings definieren. Dieser Abschnitt demonstriert das anhand einer Direktive, die das Verhalten von `ngFor` teilweise nachahmt. Dabei sei explizit darauf hingewiesen, dass dieses Beispiel lediglich der Veranschaulichung der genannten Konzepte sowie dem Nachvollziehen der prinzipiellen Funktionsweise von `ngFor` dient.

Zum Erzeugen der Direktive können Sie wie gewohnt die CLI einsetzen:

```
ng g d shared/common/repeat --export
```

Die `RepeatDirective` lässt sich abermals die aktuelle `TemplateRef` sowie die aktuelle `ViewContainerRef` injizieren (siehe Beispiel 18-24).

*Beispiel 18-24: Die RepeatDirective rendert ihr Template mehrfach.*

```
// src/app/shared/common/repeat.directive.ts

import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';

@Directive({
  selector: '[appRepeat]'
})
export class RepeatDirective {

  constructor(
    private templateRef: TemplateRef<any>,
```

```

    private viewContainer: ViewContainerRef
  ) { }

@Input() set appRepeatOf(items: any[]) {
  this.viewContainer.clear();

  let i = 0;
  for (const item of items) {

    i++;

    const context = {
      $implicit: item,
      index: i - 1
    };

    this.viewContainer.createEmbeddedView(
      this.templateRef,
      context);
  }
}

}

```

Interessant wird es im Setter, der die zu iterierenden Elemente entgegennimmt. Anders als im Beispiel aus dem letzten Abschnitt weist er nicht den Namen des vom Selektor verwendeten HTML-Attributs auf. Vielmehr ergänzt er diesen Namen um das Suffix Of (flightRepeatOf). Diese von Angular vorgesehene Konvention erlaubt später die Nutzung eines of-Operators im Template:

```

<div *appRepeat="let flight of flights">
  {{flight | json}}
</div>

```

Immer wenn per Datenbindung neue Elemente ankommen, löscht der Setter den aktuellen ViewContainer und rendert das Template für jedes Element. Dabei über gibt der Setter dem ViewContainer ein Kontextobjekt mit Informationen, die innerhalb des Templates ausgewertet werden können. Dieser Kontext kann beliebige Eigenschaften ins Template transportieren. Eine besondere Bedeutung kommt der Eigenschaft \$implicit zu. Ihren Wert weist Angular in unserem Fall der mit let definierten Variablen flights zu:

```

<div *appRepeat="let flight of flights">
  {{flight | json}}
</div>

```

Um zu verstehen, wie es dazu kommt, schauen wir uns zunächst jedoch die explizite Schreibweise an:

```

<ng-template appRepeat [appRepeatOf]="flights" let-flight let-i="index">
  <div>{{flight | json}}</div>
</ng-template>

```

Die Attribute `let-flight` und `let-i` führen für das Template zwei Variablen ein: `flight` und `i`. Die Nutzung des Präfixes `let` ist eine von Angular vorgegebene Konvention.

Die Variable `i` verweist explizit auf den Wert `index` aus dem Kontextobjekt. Im Gegensatz dazu verweist `flight` auf keinen Eintrag und erhält somit den Wert von `$implicit`.

Auch hier können Sie mit dem von Angular angebotenen syntaktischen Zucker die Schreibweise vereinfachen:

```
<div *appRepeat="let flight of flights; let i = index">
  <div>{i}: {{flight | json}}</div>
</div>
```

Das zeigt, dass Angular den an `*appRepeat` übergebenen Wert parst und in die oben gezeigte explizite Schreibweise überführt. Alle mit `let` eingeführten Variablen erhalten Werte aus dem Kontext, und Schlüsselwörter wie `of` werden zu Datenbindungen. Die dazu von Angular erlaubte Grammatik wird als Microsyntax bezeichnet.



So wie wir hier den Operator `of` durch Einrichten eines Setters mit dem Namen `flightRepeatOf` definiert haben, können Sie auch weitere Operatoren einführen. Wollten Sie beispielsweise statt `of` das Schlüsselwort `in` verwenden, bräuchten Sie ein Input `appRepeatIn`.

## Eine einfache DataTable umsetzen

Nachdem wir in den letzten Abschnitten die Theorie hinter strukturellen Direktiven betrachtet haben, folgt hier nun ein praktisches Beispiel. Wir demonstrieren die Umsetzung einer einfachen `DataTableComponent`, die ein Array mit Objekten entgegennimmt und dieses als Tabelle anzeigt (siehe Abbildung 18-5).

Upcoming Flights			
1	Hamburg	Berlin	01.02.2025 17:00
2	Hamburg	Frankfurt	01.02.2025 17:30
3	Hamburg	Mallorca	01.02.2025 17:45

Abbildung 18-5: *DataTable*

Für jede Spalte soll dazu ein Template angegeben werden können:

```
<app-data-table [data]="flights">
  <div *appTableField="let data as 'id'">{{data}}</div>
  <div *appTableField="let data as 'from'">{{data}}</div>
  <div *appTableField="let data as 'to'">{{data}}</div>
  <div *appTableField="let data as 'date'">{{data | date:'dd.MM.yyyy HH:mm'}}</div>
</app-data-table>
```

Das Array mit den darzustellenden Objekten bindet das Beispiel an data.

Bei den Templates handelt es sich um Elemente mit einem \*appTableField-Attribut. Dahinter verbirgt sich eine strukturelle Direktive. Mit der Microsyntax kann der Aufrufer festlegen, welche Objekteigenschaft das Template präsentiert.

Zum Generieren des Grundgerüsts unserer Direktive lässt sich die CLI verwenden:

```
ng generate directive shared/controls/data-table/table-field --export
```

Vergewissern Sie sich auch hier, dass diese Direktive vom Modul der Wahl deklariert und exportiert wird. In unserem Fall sollte die CLI die nötigen Einträge im SharedModule einrichten (siehe Beispiel 18-25).

*Beispiel 18-25: TableFieldDirective bei SharedModule registrieren*

```
// src/app/shared/shared.module.ts
[...]
import { TableFieldDirective } from './controls/data-table/table-field.directive';

@NgModule({
  imports: [
    CommonModule,
    FormsModule,
  ],
  declarations: [
    [...]
    TableFieldDirective
  ],
  exports: [
    [...]
    TableFieldDirective
  ]
})
export class SharedModule { }
```

Die Implementierung der Direktive stellt ihre TemplateRef als öffentliche Eigenschaft zur Verfügung (siehe Beispiel 18-26).

*Beispiel 18-26: Implementierung der TableFieldDirective*

```
// src/app/shared/controls/data-table/table-field.directive.ts

import { Directive, Input, TemplateRef } from '@angular/core';

@Directive({
  selector: '[appTableField]'
})
export class TableFieldDirective {

  @Input('appTableFieldAs') propName = '';

  constructor(public templateRef: TemplateRef<any>) { }
}
```

Diese öffentliche TemplateRef erlaubt anderen Komponenten, das Template der Direktive zu rendern.

Das @Input ist der Schlüssel zur Nutzung der Microsyntax. Die damit versehene Eigenschaft nennt sich zwar propName, allerdings veröffentlicht der Input-Dekorator sie unter dem Namen appTableFieldAs. Das entspricht dem Selektor der Direktive erweitert um das Suffix As. Gemäß den oben diskutierten Regeln rund um die Microsyntax erhält diese Eigenschaft jenen Wert, der dem Schüsselwort as nachfolgt. Im Fall von

```
<div *appTableField="let data as 'id'>[...]</div>
```

ist das der String *id*.

Die DataTableComponent lässt sich mit dem folgenden CLI-Befehl generieren:

```
ng g c shared/controls/data-table --export
```

Wie immer sollten Sie sich auch hier vergewissern, dass die neue Komponente korrekt deklariert und exportiert wird (siehe Beispiel 18-27).

*Beispiel 18-27: DataTableComponent im SharedModule importieren, deklarieren und exportieren*

```
// src/app/shared/shared.module.ts

[...]
import { DataTableComponent } from './controls/data-table/data-table.component';

@NgModule({
  imports: [
    CommonModule,
    FormsModule,
  ],
  declarations: [
    [...]
    DataTableComponent
  ],
  exports: [
    [...]
    DataTableComponent
  ]
})
export class SharedModule { }
```

Die Implementierung der Komponente findet sich in Beispiel 18-28.

*Beispiel 18-28: Implementierung der DataTableComponent*

```
// src/app/shared/controls/data-table/data-table.component.ts
```

```
import { Component, ContentChildren, Input, OnInit, QueryList } from '@angular/core';
import { TableFieldDirective } from './table-field.directive';
```

```
@Component({
```

```

    selector: 'app-data-table',
    templateUrl: './data-table.component.html',
    styleUrls: ['./data-table.component.scss']
})
export class DataTableComponent implements OnInit {

  @ContentChildren(TableFieldDirective)
  fields: QueryList<TableFieldDirective> | undefined;

  get fieldList() {
    return this.fields?.toArray();
  }

  @Input() data: Array<any> = [];

  constructor() { }

  ngOnInit(): void {
  }
}

```

Der Dekorator ContentChildren weist fields sämtliche beim Aufruf übergebenen Elemente mit einer TableFieldDirective zu. Da ContentChildren eine Auflistung vom Typ QueryList liefert, muss fields für die Nutzung im Template in ein Array umgewandelt werden. Der Getter fieldList kümmert sich darum.

Der Input data nimmt das Array mit den anzugezeigenden Objekten entgegen.

Das Template iteriert die Einträge im data-Array und erzeugt mit jedem Eintrag eine Tabellenzeile (siehe Beispiel 18-29).

#### *Beispiel 18-29: Template der DataTableComponent*

```

<!-- src/app/shared/controls/data-table/data-table.component.html -->

<table class="table">

  <tr *ngFor="let row of data">

    <td *ngFor="let f of fieldList">
      <ng-container *ngTemplateOutlet="f.templateRef; context: { $implicit: row[f.propName] }"></ng-container>
    </td>

  </tr>

</table>

```

Pro Eintrag iteriert das Template außerdem alle Einträge der fieldList und generiert jeweils eine Zelle. Die Zelle zeigt daraufhin das zugewiesene Template mit der von Angular gelieferten ngTemplateOutlet-Direktive an.

Ihre Microsyntax nimmt die zu nutzende TemplateRef entgegen. Der zusätzlich angegebene context definiert Daten, die innerhalb des Templates zur Verfügung stehen. Wie unter »Microsyntax« auf Seite 445 erwähnt, erhält \$implicit den innerhalb der Direktive verwendeten Standardwert. Bei einem Aufruf von

```
<div *appTableField="let data as 'id'">[...]</div>
```

weist Angular diesen Wert data zu.

Um die DataTableComponent auszuprobieren, erweitern wir die BookingHistoryComponent um ein Array mit Flugbuchungen (siehe Beispiel 18-30).

*Beispiel 18-30: Die BookingHistoryComponent wird um ein Array flights erweitert.*

```
// src/app/customer/booking-history/booking-history.component.ts

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

// Import hinzufügen:
import { Flight } from 'src/app/flight-booking/flight';

@Component({
  selector: 'app-booking-history',
  templateUrl: './booking-history.component.html',
  styleUrls: ['./booking-history.component.scss']
})
export class BookingHistoryComponent implements OnInit {

  // Eigenschaft hinzufügen
  flights: Flight[] = [
    { id: 1, from: 'Hamburg', to: 'Berlin', date: '2025-02-01T17:00+01:00' },
    { id: 2, from: 'Hamburg', to: 'Frankfurt', date: '2025-02-01T17:30+01:00' },
    { id: 3, from: 'Hamburg', to: 'Mallorca', date: '2025-02-01T17:45+01:00' }
  ];

  constructor() { }

  ngOnInit(): void {
  }
}
```

Anschließend platzieren wir im Template den Aufruf der DataTableComponent und binden sie an das flights-Array (siehe Beispiel 18-31).

*Beispiel 18-31: Aufruf der DataTableComponent*

```
<!-- src/app/customer/booking-history/booking-history.component.html -->

<h1>Booking History</h1>

<app-tabbed-pane>
  <app-tab title="Upcoming Flights">
    <app-data-table [data]="flights">
      <div *appTableField="let data as 'id'">{{data}}</div>
```

```

<div *appTableField="let data as 'from'">{{data}}</div>
<div *appTableField="let data as 'to'">{{data}}</div>
<div *appTableField="let data as 'date'">
    {{data | date:'dd.MM.yyyy HH:mm'}}</div>
</app-data-table>
</app-tab>
[...]
</app-tabbed-pane>

```

Falls Sie sämtliche Beispiele dieses Kapitels nachvollzogen haben, können Sie diesen Aufruf wie wir innerhalb des ersten eingerichteten Registerblatts stattfinden lassen.

## **ViewContainerRef direkt zum Einblenden von Templates verwenden**

Im letzten Abschnitt haben wir die von Angular gelieferte `ngTemplateOutlet`-Direktive eingesetzt, um ein Template darzustellen. Hierbei handelt es sich lediglich um einen Wrapper für die `ViewContainerRef`. Damit repräsentiert Angular jenen Abschnitt, in dem es ein Template (oder eine ganze Komponente) einfügt.

Für komplexere Szenarien lässt sich die `ViewContainerRef` auch direkt verwenden. Zur Veranschaulichung zeigt Beispiel 18-32 eine eigene Implementierung von `ngTemplateOutlet`.

*Beispiel 18-32: Eine benutzerdefinierte TemplateOutletDirective*

```

import { Directive, Input, OnInit, TemplateRef, ViewContainerRef } from '@angular/core';

@Directive({
  selector: '[appCustomTemplateOutlet]'
})
export class CustomTemplateOutletDirective implements OnInit {

  @Input('appCustomTemplateOutlet') template: TemplateRef<any> | undefined;
  @Input('appCustomTemplateOutletContext') context: any;

  constructor(private viewContainer: ViewContainerRef) { }

  ngOnInit(): void {
    if (!this.template) {
      return;
    }
    this.viewContainer.clear();

    this.viewContainer.createEmbeddedView(this.template, this.context);

    const ref = this.viewContainer.createEmbeddedView(this.template, this.context);
    const nativeElement = ref.rootNodes.pop();
  }
}

```

Diese Direktive holt sich per Dependency Injection die ViewContainerRef des Hosts. Über die Datenbindung erhält sie die dort einzufügende TemplateRef sowie das context-Objekt. Das Template fügt es mit der Methode createEmbeddedView in den ViewContainer ein.

Diese Methode fügt jedoch nur das Template an. Ein eventuell schon eingefügtes Template bleibt unberührt. Da wir das nicht wollen, rufen wir zuvor die Methode clear auf.

Das von createEmbeddedView zurückgelieferte Objekt gibt uns außerdem Zugriff auf das eingefügte Template. Ihre Eigenschaft rootNodes liefert sämtliche Root-Elemente des Templates. Im Fall von

```
<div *appTableField="let data as 'id'">{{data}}</div>
```

ist das div selbst das Root-Element.

In diesen Fällen existiert immer genau ein Root-Element. Verwenden wir jedoch den von strukturellen Direktiven gebotenen Syntaxzucker nicht, sondern ng-template, kann es hingegen mehrere Root-Elemente geben:

```
<ng-template>
  <div>Erstes Root-Element</div>
  <div>Zweites Root-Element</div>
</ng-template>
```

## Bestehende ViewContainer ergänzen

Komponentenbibliotheken müssen häufig bestehende ViewContainer ergänzen. Denken Sie zum Beispiel an ein Eingabefeld, das dynamisch um ein paar Tags für einen Hilfetext, der bei einem Mouse-over angezeigt wird oder auf einen Validierungsfehler aufmerksam macht:

```
<input [appInfo]="info">

<ng-template #info>
  <p>Important information!</p>
  <p>Be <strong>careful</strong> when filling this out!</p>
</ng-template>
```

Der ViewContainer von appInfo beinhaltet hier ein einziges input-Element. Allerdings können wir ihn um weitere Templates erweitern. Dazu nimmt die Direktive ein Template per Property-Binding entgegen (siehe Beispiel 18-33).

### Beispiel 18-33: Implementierung der InfoDirective

```
// src/app/shared/controls/info.directive.ts
```

```
import { Directive, Input, OnInit, TemplateRef, ViewContainerRef } from '@angular/core';

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

```

export class InfoDirective implements OnInit {
  @Input('appInfo') template: TemplateRef<any> | undefined;
  constructor(private viewContainer: ViewContainerRef) {
  }
  ngOnInit(): void {
    if (!this.template) {
      return;
    }
    const ref = this.viewContainer.createEmbeddedView(this.template);
    ref.rootNodes.forEach(nativeElement => {
      nativeElement.style.color = 'red';
    });
  }
}

```

Da `createEmbeddedView` den `ViewContainer` lediglich ergänzt, erscheint das Template hier unter dem `input`. Das zurückgelieferte Objekt gibt Zugriff auf die Root-Knoten des Templates. Im hier betrachteten Beispiel handelt es sich um die beiden `p`-Elemente, nicht jedoch um das im zweiten `p`-Element platzierte `strong`-Element.

Für diese Root-Knoten bietet Angular keine Abstraktion mehr. Es handelt sich um native DOM-Knoten. Damit können wir direkt interagieren. Das Beispiel demonstriert das, indem es die Schriftfarbe auf red setzt.

## Dialoge dynamisch einblenden

Neben der `ngTemplateOutlet`-Direktive zum Einblenden von Templates bietet Angular auch eine `ngComponentOutlet`-Direktive, die ein dynamisches Erzeugen von Komponenten erlaubt:

```
<ng-container *ngComponentOutlet="comp; injector: injector"></ng-container>
```

Bei `comp` handelt es sich um den Typ einer Komponente, und der `injector` stellt die zu injizierenden Services bereit.

In diesem Abschnitt möchten wir basierend darauf ein weiteres Beispiel für den Einsatz struktureller Direktiven zeigen. Dabei handelt es sich um einen Dialog Service, der es erlaubt, beliebige Komponenten als Dialoge einzublenden. Diese Komponenten werden bei Bedarf in einem `ngComponentOutlet` erzeugt (siehe Abbildung 18-6).

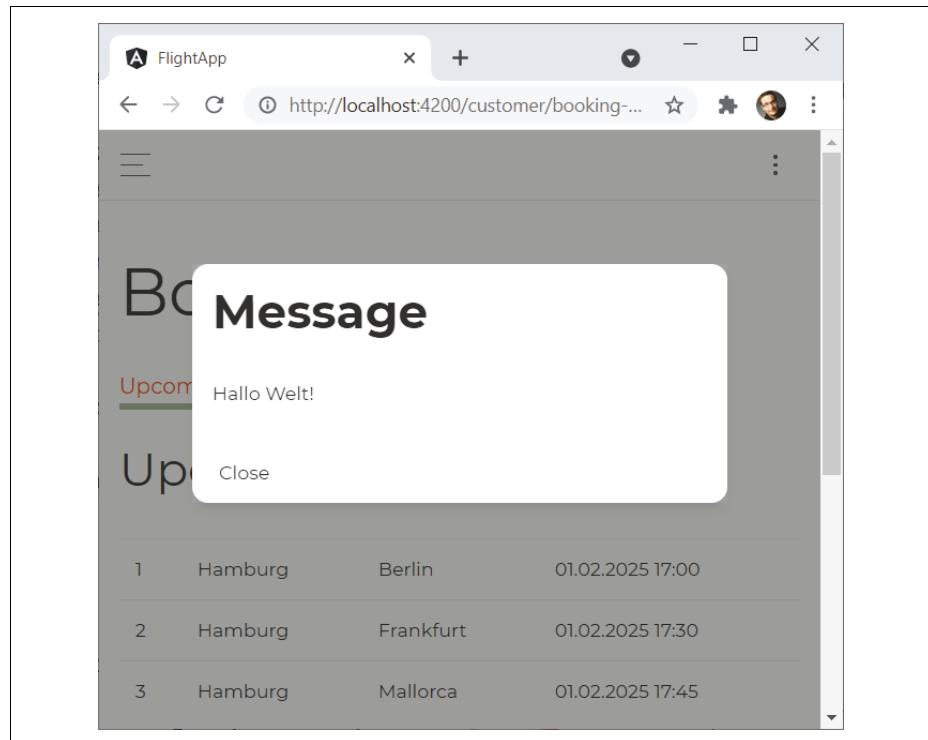


Das vom Angular-Team bereitgestellte *Angular Component Development Kit* (CDK) bietet für diesen Zweck das Konzept von *Overlays* und *Portals* (<https://material.angular.io/cdk/overlay/overview>).

Beispiel 18-34 veranschaulicht die Nutzung des `DialogService` im Kontext der oben eingerichteten `BookingHistoryComponent`.

*Beispiel 18-34: DialogService einbinden*

```
// src/app/customer/booking-history/booking-history.component.ts  
[...]  
  
import { DemoDialogComponent }  
    from 'src/app/shared/controls/dialog/demo-dialog/demo-dialog.component';  
  
import { DialogService } from 'src/app/shared/controls/dialog/dialog.service';  
  
@Component([...])  
export class BookingHistoryComponent implements OnInit {  
  
    [...]  
  
    constructor(private dialogService: DialogService) {  
    }  
  
    ngOnInit(): void {  
    }  
  
    showDialog(): void {  
        this.dialogService.show(DemoDialogComponent, 'Hallo Welt!');  
    }  
}
```



*Abbildung 18-6: Eingeblendeter Dialog*

Die vom DialogService angebotene show-Methode nimmt den Typ jener Komponente, die als Dialog angezeigt werden soll, entgegen. Beim zweiten Parameter handelt es sich um eine beliebige Information, die dieser Komponente übergeben wird.

Die Idee hinter der hier beschriebenen Implementierung sieht vor, dass die show-Methode die übergebenen Informationen an ein DialogOutlet weiterleitet. Dabei handelt es sich um eine Komponente, die als Platzhalter fungiert. Sie wird an einer beliebigen Stelle platziert und erzeugt bei Bedarf die gewünschte Dialogkomponente.

Für den DialogService benötigen wir ein paar Hilfskonstrukte. Das Interface DialogInfo repräsentiert die Daten, die der DialogService an das DialogOutlet weiterleitet:

```
// src/app/shared/controls/dialog/dialog-info.ts

import { Type } from '@angular/core';

export interface DialogInfo {
  component: Type<any> | null;
  data: any;
}
```

Der Datentyp Type<any> verweist hier auf einen beliebigen Typ. In unserem Fall handelt es sich dabei um die als Dialog anzuseigende Komponente.

Um Informationen an die Komponente weiterzuleiten, kommt Dependency Injection zum Einsatz. Hierfür definieren wir ein Token DIALOG\_DATA:

```
// src/app/shared/controls/dialog/dialog.token.ts

import { InjectionToken } from '@angular/core';

export const DIALOG_DATA = new InjectionToken<any>('DIALOG_DATA');
```

Der DialogService lässt sich mit der CLI wie folgt erzeugen:

```
ng generate service shared/controls/dialog/dialog
```

Beispiel 18-35 zeigt die Implementierung dieses Service.

*Beispiel 18-35: Implementierung des DialogService*

```
// src/app/shared/controls/dialog/dialog.service.ts

import { Injectable, Type } from '@angular/core';
import { BehaviorSubject } from 'rxjs';
import { DialogInfo } from './dialog-info';

@Injectable({
  providedIn: 'root'
})
export class DialogService {

  dialogInfo = new BehaviorSubject<DialogInfo>({
    component: null,
    data: null
  })
```

```

    });

constructor() { }

show(comp: Type<any>, data: any): void {
  this.dialogInfo.next({
    component: comp,
    data
  });
}

hide(): void {
  this.dialogInfo.next({
    component: null,
    data: null
  });
}
}

```

Die Methode `show` veröffentlicht die übergebenen Daten über ein `BehaviorSubject`. Zur Vereinfachung veröffentlicht dieses Beispiel dieses `BehaviorSubject`. Stabiler wäre die in Kapitel 11 diskutierte Vorgehensweise, die die Kombination eines privaten Subjects und eines öffentlichen (`readonly`) Observables vorsieht.

Um den Dialog wieder auszublenden, veröffentlicht die Methode `hide` lediglich `null`-Werte.

Die `DialogOutletComponent` lässt sich mit der CLI generieren:

```
ng generate component shared/controls/dialog/dialog-outlet --export
```

Die `DialogOutletComponent` nimmt die über das `BehaviorSubject` veröffentlichten Informationen entgegen (siehe Beispiel 18-36).

#### *Beispiel 18-36: Implementierung der DialogOutletComponent*

```
// src/app/shared/controls/dialog/dialog-outlet/dialog-outlet.component.ts
```

```

import { Component, Injector, OnInit, Type } from '@angular/core';
import { DialogService } from '../dialog.service';
import { DIALOG_DATA } from '../dialog.token';

@Component({
  selector: 'app-dialog-outlet',
  templateUrl: './dialog-outlet.component.html',
  styleUrls: ['./dialog-outlet.component.scss']
})
export class DialogOutletComponent implements OnInit {

  comp: Type<any> | null = null;
  injector: Injector | null = null;

  constructor(
    private dialogService: DialogService,
    private parentInjector: Injector,
  ) { }
}

```

```

ngOnInit(): void {
  this.dialogService.dialogInfo.subscribe(info => {
    // Komponententyp über Eigenschaft comp bereitstellen:
    this.comp = info.component;

    // Injector für empfangene Daten bereitstellen:
    this.injector = Injector.create({
      providers: [
        { provide: DIALOG_DATA, useValue: info.data }
      ],
      parent: this.parentInjector
    });
  });
}

```

Die DialogOutletComponent platziert den empfangenen Komponententyp in der Eigenschaft `comp`. Für die empfangenen Daten, die es an die Dialogkomponente weiterzuleiten gilt, erzeugt sie einen Injector. Dieser bietet einen Serviceprovider für das Token `DIALOG_DATA` an. Unter diesem Token veröffentlicht er die empfangenen Daten.

Außerdem verweist der Injector über die Eigenschaft `parent` auf den Injector der DialogOutletComponent. Somit kann er auch die von diesem Parent-Injector bereitgestellten Services anbieten. Dabei handelt es sich in erster Linie um sämtliche global eingerichteten Services (`providedIn: root`).

Sowohl der Komponententyp als auch der Injector werden im Template der DialogOutletComponent an die `ngComponentOutlet`-Direktive weitergegeben.

#### *Beispiel 18-37: Template der DialogOutletComponent*

```

// src/app/shared/controls/dialog/dialog-outlet/dialog-outlet.component.html

<div *ngIf="comp && injector" class="container">
  <div class="background"></div>

  <div class="dialog">
    <ng-container *ngComponentOutlet="comp; injector: injector"></ng-container>
  </div>
</div>

```

Damit die hier eingeblendete Komponente als Dialog präsentiert wird, benötigen wir ein paar Styles (siehe Beispiel 18-38).

*Beispiel 18-38: Styles der DialogOutletComponent*

```
/* src/app/shared/controls/dialog/dialog-outlet/dialog-outlet.component.scss */

.background {
    opacity: 0.4;
    background-color: black;
    left: 0px;
    top: 0px;
    width: 100%;
    height: 100%;
    position: fixed;
    z-index: 10000;
}

.container {
    left: 0px;
    top: 0px;
    width: 100%;
    height: 100%;
    position: absolute;
    z-index: 11000;
}

.dialog {
    background-color: white;
    margin-top: 100px;
    margin-left: auto;
    margin-right: auto;
    width: 400px;
}
```

Die Klasse `background` definiert einen halb transparenten Hintergrund, und das Zusammenspiel aus `container` und `dialog` platziert die Komponente horizontal zentriert über allen anderen.

Nun ist es an der Zeit, unsere Implementierung auszuprobieren. Wir benötigen dazu eine Komponente, die wir als Dialog anzeigen wollen. Diese lässt sich mit der CLI generieren:

```
ng generate component shared/controls/dialog/demo-dialog --export
```

Die Implementierung dieser Komponente findet sich in Beispiel 18-39.

*Beispiel 18-39: Implementierung von DemoDialogComponent*

```
// src/app/shared/controls/dialog/demo-dialog/demo-dialog.component.ts

import { Component, Inject, OnInit } from '@angular/core';
import { DialogService } from '../dialog.service';
import { DIALOG_DATA } from '../dialog.token';

@Component({
    selector: 'app-demo-dialog',
    templateUrl: './demo-dialog.component.html',
    styleUrls: ['./demo-dialog.component.scss']
```

```

})
export class DemoDialogComponent implements OnInit {

  constructor(
    @Inject(DIALOG_DATA) public data: string,
    private dialogService: DialogService) { }

  ngOnInit(): void {
  }

  close(): void {
    this.dialogService.hide();
  }
}

```

Die `DemoDialogComponent` lässt sich die unter dem Token `DIALOG_DATA` bereitgestellten Informationen sowie den `DialogService` injizieren. Letzteren greift ihre `close`-Methode auf, um sich selbst wieder auszublenden.

Das Template der `DemoDialogComponent` zeigt die empfangenen Daten an und bietet eine Schaltfläche, die an die Methode `close` gebunden ist (siehe Beispiel 18-40).

*Beispiel 18-40: Template der `DemoDialogComponent`*

```
<!-- src/app/shared/controls/dialog/demo-dialog/demo-dialog.component.html -->

<div class="card" style="z-index: 100000;">

  <div class="card-body">
    <h2 class="title">Message</h2>
    <p>{{data}}</p>
  </div>

  <div class="card-footer">
    <a (click)="close()">Close</a>
  </div>

</div>
```

Um die `DemoDialogComponent` einzublenden, erweitern wir die `BookingHistoryComponent`, wie in Beispiel 18-34 gezeigt.

Ihr Template erhält außerdem eine Schaltfläche, die sich an `showDialog` bindet:

```
<!-- src/app/customer/booking-history/booking-history.component.html -->
[...]

<button class="btn" (click)="showDialog()">Show Dialog</button>

<!-- Element hinzufügen -->
<app-dialog-outlet></app-dialog-outlet>
```

Außerdem ruft das Template die `DialogOutletComponent` auf. Da wir über einen Service damit kommunizieren, könnten wir sie jedoch auch an einer zentraleren Stelle, z.B. im Template der `AppComponent`, platzieren.

Klicken Sie nun auf die Schaltfläche Show Dialog, sollte sich das in Abbildung 18-7 dargestellte Ergebnis zeigen.

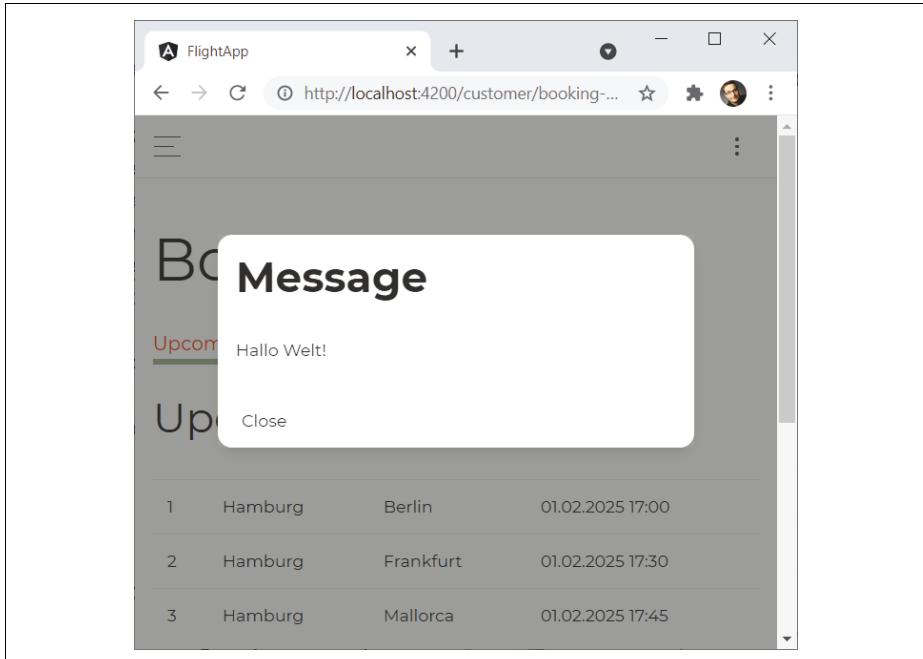


Abbildung 18-7: Eingeblendeter Dialog



Damit das hier gezeigte Beispiel funktioniert, müssen Sie an einer beliebigen Stelle Ihrer Anwendung ein `app-dialog-outlet`-Element einfügen. Um den Komfort zu erhöhen, können Sie ein paar interne Angular-Konstrukte nutzen, um diese Aufgabe zu automatisieren:

```
@NgModule({ ... })
export class SharedModule {

    constructor(
        cvr: ComponentFactoryResolver,
        injector: Injector,
        appRef: ApplicationRef,
        @Inject(DOCUMENT) document: Document
    ) {
        const factory = cvr.resolveComponentFactory
        (DialogOutletComponent);
        const compRef = factory.create(injector);

        // Attach component's view so that Angular does
        // change detection
        appRef.attachView(compRef.hostView);
    }
}
```

```

        // Add component's DOM node to the view
        document.body.appendChild(
            compRef.location.nativeElement);
    }

}

```

Den Code zum dynamischen Einfügen eines app-dialog-outlet platziert dieses Beispiel im Konstruktor des SharedModule. Per Dependency Injection lassen sich hier die benötigten Objekte beziehen. Die View der Komponente (hostView) ist der ApplicationRef der Anwendung hinzuzufügen. Das ist notwendig, damit Angular die Komponente bei der Change Detection berücksichtigt.

Außerdem ist ein DOM-Knoten (nativeElement), der die Komponente repräsentiert, in die aktuelle Seite einzufügen. Diese wird durch das document-Objekt repräsentiert. Idealerweise greifen Sie nicht direkt auf die globale Variable document zu, sondern fordern es stattdessen via Dependency Injection an. Hierzu stellt Angular das InjectionToken DOCUMENT zur Verfügung. Somit können Sie dieses Objekt später austauschen – zum Beispiel für Unit-Tests, aber auch beim serverseitigen Vorrendern von Seiten.

## ViewContainerRef direkt zum dynamischen Erzeugen von Komponenten verwenden

Im letzten Abschnitt haben wir die von Angular gebotene \*componentOutlet-Direktive verwendet, um eine Komponente dynamisch zu erzeugen. Das ist auch die von uns empfohlene Vorgehensweise. Zur Schaffung eines besseren Verständnisses dafür, wie Angular an dieser Stelle funktioniert, diskutieren wir hier eine einfache eigene Implementierung davon.

Wir haben sie mit der CLI generiert:

```
ng g d shared/controls/dialog/custom-component-outlet --export
```

Da es sich auch hier um eine strukturelle Direktive handelt, korrelieren die Namen der Inputs mit dem Selektor der Direktive (siehe Beispiel 18-41).

*Beispiel 18-41: Benutzerdefinierte Implementierung von der Direktive component-outlet*

```
// src/app/shared/controls/dialog/custom-component-outlet.directive.ts

import {
    ComponentFactoryResolver,
    Directive,
    Injector,
    Input,
    OnChanges,
    SimpleChanges,
    Type,
    ViewContainerRef
}
from '@angular/core';
```

```

@Directive({
  selector: '[appCustomComponentOutlet]'
})
export class CustomComponentOutletDirective implements OnChanges {

  @Input('appCustomComponentOutlet') component: Type<any> | undefined;

  @Input('appCustomComponentOutletInjector') injector: Injector | undefined;

  constructor(
    private cfr: ComponentFactoryResolver,
    private viewContainer: ViewContainerRef) { }

  ngOnChanges(changes: SimpleChanges): void {
    if (!this.component) {
      return;
    }
    const factory = this.cfr.resolveComponentFactory(this.component);
    const compRef = this.viewContainer.createComponent(
      factory, undefined, this.injector);
  }
}

```

Da diese Direktive die Komponentenklasse – und nicht etwa eine Instanz davon – entgegennimmt, ist die Eigenschaft `component` vom Typ `Type<any>`. Aus den weiter oben diskutierten Gründen brauchen wir eine Factory für die übergebene Komponente. Diese rufen wir mit dem injizierten `ComponentFactoryResolver` ab.

Die Methode `createComponent` erzeugt eine Instanz der Komponente und fügt sie im `ViewContainer` der Direktive ein. Das zweite Argument definiert die Position innerhalb des `ViewContainer`. Das ist interessant, wenn Sie mehrere Komponenten hintereinander einfügen. Der Wert `undefined` legt fest, dass die Komponente am Ende des `ViewContainer` einzufügen ist.

Die zurückgelieferte `ComponentReference` `compRef` gibt uns in weiterer Folge direkten Zugriff auf die erzeugte Komponente:

```

const instance = compRef.instance;
instance.someProp = 42;
instance.someEvent.subscribe( (e: unknown) => console.debug('e', e));
instance.someMethod();

```

In diesem Beispiel gehen wir davon aus, dass die Komponente `someProp`, `someEvent` und `someMethod` anbietet.

## Komponenten für Formularfelder

Sowohl beim Einsatz von Template-getriebenen Formularen als auch bei ihren reaktiven Gegenstücken muss Angular wissen, wie die jeweiligen Werte an die gewünschten Steuerelementen zu binden sind. Bei Textfeldern ist zum Beispiel die Eigenschaft `value` zu setzen, bei Checkboxen hingegen `checked`, und bei Radiobuttons sowie

Drop-down-Feldern ist der betroffene Eintrag auszuwählen. Für diese Aufgabe nutzt das Framework sogenannte Value-Accessoren. Für die von HTML standardmäßig angebotenen Steuerelemente besitzt Angular entsprechende Value-Accessor-Implementierungen, und für darüber hinausgehende Anforderungen kann eine Anwendung auch eigene Implementierungen bereitstellen.

Auf diese Weise lassen sich eigene Steuerelemente gemeinsam mit Angular-Formularen nutzen. Daneben kann die Anwendung damit das Datenbindungsverhalten von Angular anpassen, um zum Beispiel Werte im Zuge der Datenbindung zu formatieren. Dieser Abschnitt demonstriert beide Möglichkeiten anhand von Beispielen.

## Ausgaben formatieren und Eingaben parsen

Für die Umsetzung von Value-Accessoren stellt Angular das Interface `ControlValueAccessor` zur Verfügung (siehe Abbildung 18-8).

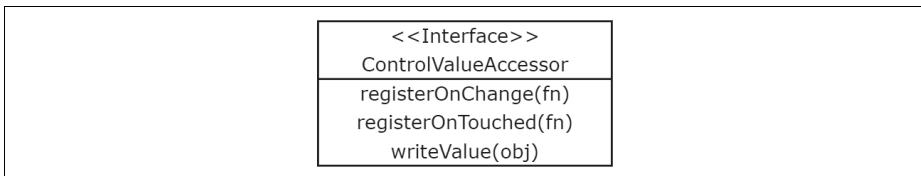


Abbildung 18-8: `ControlValueAccessor`

Die Methoden `registerOnChange` sowie `registerOnTouched` bekommen von Angular Callbacks übergeben. Immer wenn der Benutzer Daten im Steuerelement ändert, muss der Value-Accessor den an `registerOnChange` übergebenen Callback aufrufen. Erhält ein Steuerelement den Fokus, muss der Value-Accessor hingegen den an `registerOnTouched` übergebenen Callback anstoßen. Auf diese Weise informiert der Value-Accessor Angular über Benutzerinteraktionen.

Daneben gibt das Interface die Methode `writeValue` vor. Angular übergibt an diese Methode jene Werte, die das Steuerelement anzeigen soll. Auf diese Weise kann Angular das Model, zum Beispiel ein Flugobjekt, mit Steuerelementen abgleichen.

Abbildung 18-9 veranschaulicht das Zusammenspiel zwischen `ControlValueAccessor` und Angular:

1. Angular übergibt Callbacks an `registerOnChange` und `registerOnTouched`.
2. Die `ControlValueAccessor`-Implementierung hinterlegt die Callbacks in Eigenschaften.
3. Angular übergibt neue Werte an `writeValue`.
4. Die `ControlValueAccessor`-Implementierung informiert Angular über Änderungen, indem sie die Callbacks aufruft.

Das folgende Beispiel demonstriert, wie eine Anwendung damit die gebundenen Daten im Rahmen der Datenbindung modifizieren kann. Dazu wandeln wir ein

Datum, das im Model als ISO-String vorliegt, in ein deutsches Datum um und schreiben Modifikationen in Form eines ISO-Strings ins Model zurück (siehe Abbildung 18-10).

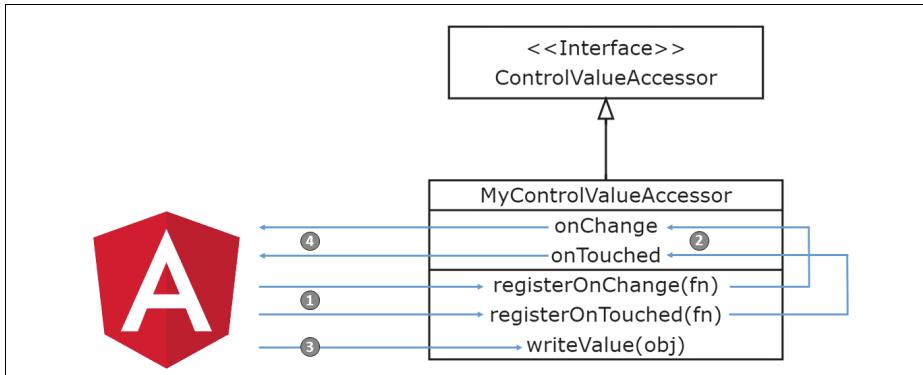


Abbildung 18-9: Nutzung eines ControlValueAccessor



Abbildung 18-10: Ein Value-Accessor übernimmt das Formatieren des Datums.

Bei diesem Beispiel handelt es sich um eine Direktive, die das Interface ControlValueAccessor implementiert (siehe Beispiel 18-42).

#### Beispiel 18-42: ControlValueAccessor zum Formatieren von Datumswerten

```

// src/app/shared/value-accessors/date-value-accessor.directive.ts

import { Directive, ElementRef, HostListener } from '@angular/core';
import { ControlValueAccessor, NG_VALUE_ACCESSOR } from '@angular/forms';

@Directive({
  // Der ControlValueAccessor gilt für sämtliche
  // input-Elemente mit einem appDate-Attribut,
  // z. B. <input appDate name="date" [(ngModel)]="date">
  selector: 'input[appDate]',

  // Die Direktive muss bei Angular unter dem Token
  // NG_VALUE_ACCESSOR registriert werden.
  providers: [
    {
      provide: NG_VALUE_ACCESSOR,
      useExisting: DateValueAccessorDirective,
      multi: true
    }
  ]
})
export class DateValueAccessorDirective implements ControlValueAccessor {

  // ElementRef repräsentiert das Eingabefeld.
  constructor(private elementRef: ElementRef) { }

  // ...
}
  
```

```

onChange = (_: any) => { };
onTouched = () => { };

// Von Angular erhaltene Daten ins Eingabefeld schreiben.
writeValue(value: string): void {

    // Formatieren:
    if (value) {
        const date = new Date(value);
        value = date.getDate() + '.'
            + (date.getMonth() + 1) + '.'
            + date.getFullYear();
    }

    const formatted = (value) ? value : '';

    // Zurückmelden:
    this.elementRef.nativeElement.value = formatted;
}

// Von Angular erhaltenen Callback
// in Eigenschaft onChange ablegen:
registerOnChange(fn: any): void {
    this.onChange = fn;
}

// Von Angular erhaltenen Callback
// in Eigenschaft onTouched ablegen:
registerOnTouched(fn: any): void {
    this.onTouched = fn;
}

// Beim Blur-Event Angular
// mit onTouched informieren:
@HostListener('blur')
blur(): void {
    this.onTouched();
}

// Beim Input-Event Angular mit onTouched
// über geänderten Wert informieren:
@HostListener('input', ['$event.target.value'])
input(value: string): void {

    // Formatieren:
    if (value) {
        const parts = value.split(/\./);
        value = parts[2] + '-' + parts[1] + '-' + parts[0];
    }

    // Zurückmelden:
    this.onChange(value);
}
}

```

Bitte achten Sie darauf, dass diese Direktive im SharedModule sowohl zu deklarieren als auch zu exportieren ist.

Ein wichtiger Aspekt dieses Beispiels ist der eingeführte Provider. Er bindet die Direktive an das Token NG\_VALUE\_ACCESSOR. Angular nutzt das an dieses Token gebundene Objekt, um im Rahmen der Datenbindung mit dem Steuerelement zu kommunizieren.

Zum Ausprobieren können Sie die FlightSearchComponent um eine Eigenschaft date mit dem aktuellen Datum versehen (siehe Beispiel 18-43).

*Beispiel 18-43: FlightSearchComponent um Eigenschaft date erweitern*

```
// src/app/flight-search/flight-search.component.ts
```

```
[...]
export class FlightSearchComponent implements OnInit {
    [...]
    from = 'Hamburg';
    to = 'Graz';
    flights: Array<Flight> = [];
    selectedFlight: Flight | null = null;
    delayFilter = false;

    date = new Date().toISOString();
    [...]
}
```

Danach können Sie die Eigenschaft date an ein Eingabefeld binden (siehe Beispiel 18-44).

*Beispiel 18-44: Eigenschaft date binden und mit appDate formatieren*

```
<!-- src/app/flight-search/flight-search.component.html -->
<div class="form-group">
    <label>Date:</label>
    <input [(ngModel)]="date" name="date" appDate class="form-control">
    {{date}}
</div>
```

Durch Nutzung des Attributs appDate wenden Sie den ControlValueAccessor auf dieses Eingabefeld an.

## Eigene Formularsteuerelemente

Auch eigene Steuerelemente können ControlValueAccessor implementieren, damit sie mit Template-getriebenen und reaktiven Formularen zusammenspielen. Um das zu demonstrieren, erweitern wir hier das DateControl aus Kapitel 4 (siehe Abbildung 18-11).



Abbildung 18-11: Eigenes Formularsteuerelement

Durch das Implementieren von ControlValueAccessor soll das DateControl über ng Model Daten gebunden bekommen:

```
<app-date [(ngModel)]="date"></app-date>
```

Alternativ dazu kann sich solch eine Komponente jedoch auch über reaktive Formulare an ein vordefiniertes FormControl-Objekt binden:

```
<form [formGroup]="filter">
  <app-date formControlName="date"></app-date>
  [...]
</form>
```

Damit das funktioniert, muss die DateComponent selbst das Interface ControlValue Accessor implementieren. Somit kann sie das gebundene Datum von Angular empfangen und Änderungen zurückmelden (siehe Beispiel 18-45).

*Beispiel 18-45: Eigenes Steuerelement implementiert ControlValueAccessor.*

```
// src/app/shared/date/date.component.ts

import { Component } from '@angular/core';
import { ControlValueAccessor, NgControl } from '@angular/forms';

@Component({
  selector: 'app-date',
  templateUrl: './date.component.html',
  styleUrls: ['./date.component.scss']
})
export class DateComponent implements ControlValueAccessor {

  // Einzelteile des gebundenen Datums:
  day: number | null = null;
  month: number | null = null;
  year: number | null = null;
  hour: number | null = null;
  minute: number | null = null;

  constructor(private c: NgControl) {
    // Das Control ist sein eigener ControlValueAccessor.
    // Deswegen übergibt es sich selbst an Angular.
    c.valueAccessor = this;
  }

  onChange = (_: any) => { };
  onTouched = () => { };

  writeValue(dateString: string): void {
```

```

    if (!dateString) {
        return;
    }

    // Übergebenes Datum in Einzelteile erlegen.
    // Diese Einzelteile werden per Datenbindung
    // ausgegeben.
    const date = new Date(dateString);
    this.day = date.getDate();
    this.month = date.getMonth() + 1;
    this.year = date.getFullYear();
    this.hour = date.getHours();
    this.minute = date.getMinutes();
}

registerOnChange(fn: any): void {
    this.onChange = fn;
}

registerOnTouched(fn: any): void {
    this.onTouched = fn;
}

apply(): void {

    if (!this.year || !this.month || !this.day || !this.hour || !this.minute) {
        return;
    }

    // Einzelteile zu Datum zusammenfügen:
    const date = new Date(this.year, this.month - 1, this.day, this.hour,
        this.minute);

    // Datum als ISO-String zurückmelden:
    this.onChange(date.toISOString());
}
}

```

Bitte beachten Sie, dass sich das Steuerelement eine Instanz von NgControl injizieren lässt. Über dieses Objekt kann es sich selbst als ControlValueAccessor zur Verfügung stellen.

Das Steuerelement zerlegt das übergebene Datum in seine Einzelteile und präsentiert diese in jeweils eigenen Eingabefeldern. Nach dem Aktualisieren der Einzelteile fügt seine apply-Methode sie wieder zusammen und meldet sie an Angular zurück (siehe Beispiel 18-46).

#### *Beispiel 18-46: Template für DateComponent*

```
<!-- src/app/shared/date/date.component.html -->

<form class="form-inline">
    <input [(ngModel)]="day" name="day" maxlength="2" style="width:50px"
        class="form-control">
    .
    .
    .

```

```

<input [(ngModel)]="month" name="month" maxlength="2" style="width:50px"
       class="form-control">
.
<input [(ngModel)]="year" name="year" maxlength="4" style="width:70px"
       class="form-control">
 &nbsp;
<input [(ngModel)]="hour" name="hour" maxlength="2" style="width:50px"
       class="form-control">
:
<input [(ngModel)]="minute" name="minute" maxlength="2" style="width:50px"
       class="form-control">
 &nbsp;
<input type="button" value="Apply" (click)="apply()" class="btn btn-default">
</form>

```

Um diese Implementierung zu testen, sollten Sie – wie üblich – sicherstellen, dass die DateComponent im SharedModule deklariert und exportiert wird. Danach können Sie sie direkt in Ihrer FlightSearchComponent aufrufen:

```

<!-- src/app/flight-search/flight-search.component.html -->
[...]
<div class="form-group">
  <label>Date:</label>
  <app-date [(ngModel)]="date" name="date"></app-date>
  {{date}}
</div>
[...]

```

Hierbei gehen wir davon aus, dass Ihre FlightSearchComponent die im letzten Abschnitt verwendete Eigenschaft date aufweist.

## Zusammenfassung

Angular bietet einige Konzepte für Komponenten. Beispielsweise kann eine Komponente über Content Projection Markup entgegennehmen. Außerdem bieten Komponenten mehrere Möglichkeiten, um miteinander zu kommunizieren, darunter Handles, Services oder die Referenzierung von Parent-Komponenten. Daneben können Komponenten auch Child-Komponenten abrufen.

In Fällen, in denen Sie nur zusätzliche Logik ohne Ausgabe benötigen, bieten sich Direktiven an. Neben den einfachen attributbasierten Direktiven können Sie in Angular auch die sogenannten strukturellen Direktiven einsetzen. Beispiele dafür sind ngIf und ngFor, die das Framework ab Werk bietet. Strukturelle Direktiven haben immer ein Template und können es bei Bedarf einmal oder mehrfach rendern. Dabei übergeben sie Kontextinformationen, die das Template im Rahmen der Datenbindung aufgreifen kann.

# Wiederverwendbare Bibliotheken und Monorepos

Bis jetzt hat unser Angular-Projekt lediglich aus einer Anwendung bestanden. Die CLI erlaubt es allerdings, ein Projekt in mehrere wiederverwendbare Bibliotheken und Anwendungen zu unterteilen. Da das gesamte Projekt für gewöhnlich in einem Quellcode-Repository verwaltet wird und nun aus mehreren Anwendungen und Bibliotheken besteht, ist hierbei auch von einem Monorepo die Rede. Eine alternative Bezeichnung ist Workspace.

Die Bibliotheken lassen sich als `npm`-Pakete bereitstellen. Somit lassen sie sich auch in anderen Projekten nutzen. In diesem Kapitel werden wir mit der CLI ein einfaches Monorepo mit einer Logging-Bibliothek erstellen. Diese nutzen wir mit einer Demoanwendung, die zum selben Monorepo gehört. Danach stellen wir die Bibliothek über eine `npm`-Registry bereit.



Monorepos eignen sich übrigens nicht nur zu Schaffung wiederverwendbarer Bibliotheken, sondern kommen auch bei sehr großen Projekten zum Einsatz. In diesem Fall dient das Monorepo der Untergliederung der Gesamtlösung in einzelne kleinere Anwendungen und/oder Bibliotheken. Solche kleinen Subprojekte sind besser überschaubar und somit einfacher zu warten. Die Bibliotheken müssen bei dieser Spielart auch nicht via `npm` veröffentlicht werden, sondern können auch nur lokal von der Anwendung/den Anwendungen konsumiert werden.

## Monorepo erstellen

Aus Sicht der CLI ist ein Monorepo ein herkömmliches Angular-Projekt. Dieses lässt sich mit `ng new` generieren:

```
ng new logger --create-application false
```

Normalerweise generiert die CLI für ein Angular-Projekt einen Ordner `src`, der den gesamten Anwendungscode beherbergt. Da wir das Projekt jedoch in verschiedene Anwendungen und Bibliotheken unterteilen wollen, ergibt dieser zentrale `src`-Ordner keinen Sinn. Deswegen weisen wir die CLI mit dem Schalter `--create-application false` an, diesen Ordner nicht zu generieren.

Im Projektordner lassen sich nun die gewünschten Anwendungen und Bibliotheken einrichten:

```
cd logger
ng g lib logger-lib
ng g app playground-app
```

Verwenden Sie beim Generieren die von der CLI vorgeschlagenen Standardeinstellungen. Die generierten Anwendungen und Bibliotheken finden sich nach dem Generieren im Ordner *projects* (siehe Abbildung 19-1).

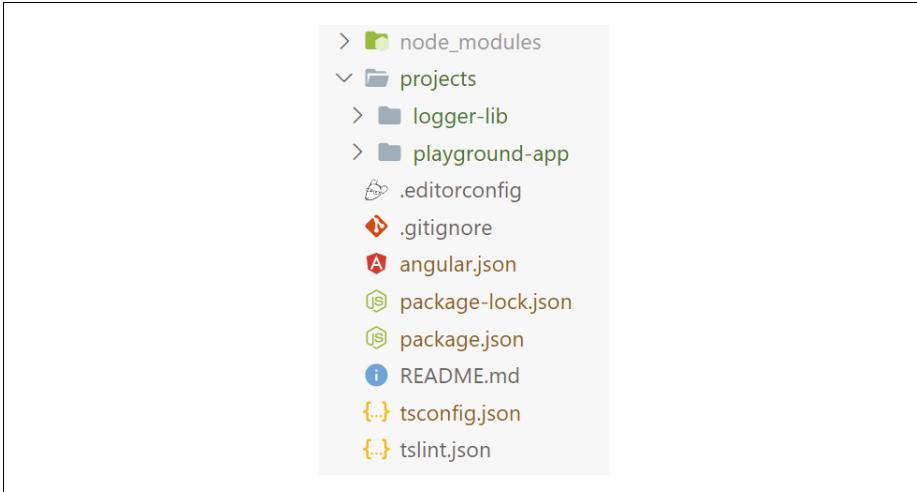


Abbildung 19-1: Aufbau eines Monorepos

Da in einem Monorepo mit Anwendungen und Bibliotheken mehrere Subprojekte vorliegen, müssen Sie bei den einzelnen CLI-Anweisungen das gewünschte Subprojekt angeben:

```
ng generate component MyComponent --project playground-app
ng generate component OtherComponent --project logger-lib

ng serve playground-app -o

ng build playground-app
ng build logger-lib
```

Eines der Subprojekte lässt sich auch als Standardprojekt konfigurieren. Dieses zieht die CLI heran, wenn bei einem Aufruf kein Projekt angegeben wird. Der Name dieses Standardprojekts findet sich in der Eigenschaft `defaultProject` am Ende der `angular.json`:

```
"defaultProject": "logger-lib"
```

Die CLI trägt hier das erste erzeugte Subprojekt ein. Sie können jedoch die Eigenschaft abändern, sodass sie auf ein anderes Projekt verweist.

Anwendungen eines Monorepos sind genau so strukturiert, wie wir es von den vorangegangenen Kapiteln gewohnt sind. Auf den Aufbau von Bibliotheken gehen wir im nächsten Abschnitt ein.

## Aufbau von Bibliotheken

Genau wie Anwendungen bestehen Bibliotheken aus TypeScript-Dateien, die unter anderem Angular-Building-Blocks wie Komponenten oder Services repräsentieren. Diese befinden sich im Unterordner `src/lib` (siehe Abbildung 19-2).

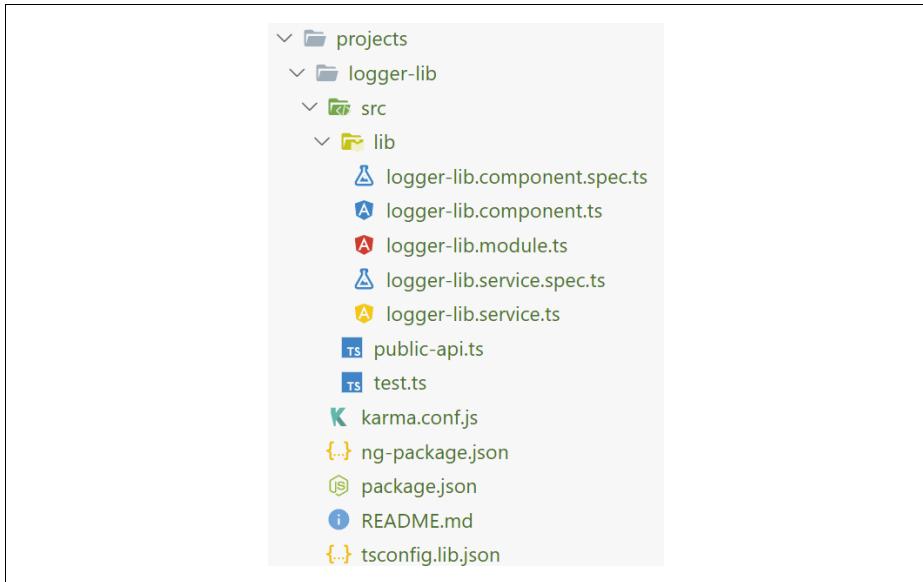


Abbildung 19-2: Aufbau einer Bibliothek in einem Monorepo

Zur Veranschaulichung generieren wir in der `logger-lib` einen neuen `LoggerService` (nicht zu verwechseln mit dem `LoggerLibService`, den die CLI beim Erzeugen der gleichnamigen Bibliothek eingerichtet hat):

```
ng g service logger --project logger-lib
```

Diesem spendieren wir eine einfache Methode `log`:

*Beispiel 19-1: Einfacher LoggerService*

```
// projects/logger-lib/src/lib/logger.service.ts

import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class LoggerService {
```

```

constructor() { }

// Hinzufügen
log(message: string): void {
  const date = new Date().toISOString().substr(0, 10);
  console.log(`[${date}] ${message}`);
}

}

```

Damit dieser Service von anderen Anwendungen genutzt werden kann, ist er in der Datei *public-api.ts* zu exportieren:

```

// projects/logger-lib/src/public-api.ts

[...]

export * from './lib/logger.service';

```

Diese Datei ist der Einsprungpunkt in unsere Bibliothek. Der Dateiname des Einsprungpunkts lässt sich übrigens in der von der CLI generierten Datei *ng-package.json* konfigurieren:

*Beispiel 19-2: Die Steuerdatei projects/logger-lib/ng-package.json*

```
{
  "$schema": "../../node_modules/ng-packagr/ng-package.schema.json",
  "dest": "../../dist/logger-lib",
  "lib": {
    "entryFile": "src/public-api.ts"
  }
}
```

Die Eigenschaft *dest* verweist übrigens auf das Verzeichnis, in dem das *npm*-Paket im Zuge des Bauens abzulegen ist.

Bevor Sie eine Bibliothek als *npm*-Paket bereitstellen, müssen Sie auch seine Metadaten in der generierten Datei *package.json* warten (siehe Beispiel 19-3).

*Beispiel 19-3: Die Datei projects/logger-lib/package.json*

```
{
  "name": "@my/logger-lib",
  "version": "0.0.1",
  "peerDependencies": {
    "@angular/common": ">= 11.0.0",
    "@angular/core": ">= 11.0.0"
  },
  "dependencies": {
    "tslib": "^2.0.0"
  }
}
```

Eine vollständige Übersicht aller hier möglichen Einstellungen findet sich unter <https://docs.npmjs.com>. Wir beschränken uns hier auf die wichtigsten Eigenschaften:

#### *name*

Name des `npm`-Pakets. Unter diesem Namen wird das Paket veröffentlicht, aber auch installiert.

#### *version*

Version des Pakets. Bei jeder einzelnen Bereitstellung muss eine noch verfügbare Versionsnummer angegeben werden.

#### *peerDependencies*

Abhängigkeiten, die der Konsument zusätzlich installieren muss, damit er dieses Paket nutzen kann. Sie können mit booleschen Ausdrücken den unterstützten Versionsbereich angeben.

#### *dependencies*

Abhängigkeiten, die gemeinsam mit dem vorliegenden Paket installiert werden.

Den Paketnamen kann ein sogenannter Scope vorangestellt werden. Diese sind mit einem @ einzuleiten. Im gezeigten Beispiel wurde zur Veranschaulichung der Scope `@my` gewählt. Scopes bringen ein wenig Ordnung ins Spiel, weil sie ersichtlich machen, aus welchem Bereich das Paket stammt. Häufig kommt der Unternehmensname und/oder der Projektname als Scope zum Einsatz. Ein weiterer Vorteil von Scopes ist, dass nur Personen mit entsprechenden Rechten etwas unter einem Scope bereitstellen dürfen.

In der Regel sind `peerDependencies` gegenüber »herkömmlichen« `dependencies` zu bevorzugen, da diese dem Konsumenten nicht eine bestimmte Version aufzwingen.



Abgesehen von der von TypeScript benötigten `tslib` müssen Sie die Nutzung weiterer `dependencies` explizit erlauben, um einen versehentlichen Einsatz zu vermeiden.

Stellen wir uns dazu den folgenden Eintrag in der `package.json` vor:

```
"dependencies": {  
    "tslib": "^2.0.0",  
    "sha-256-js": "1.0.3"  
}
```

Damit die CLI den Einsatz von `sha-256-js` als »herkömmliche« dependency erlaubt, ist dieses Paket explizit in der von der CLI generierten Datei `ng-package.json` einzutragen:

```
{  
    [...]  
    "allowedNonPeerDependencies": [  
        "sha-256-js"  
    ]  
}
```

## Bibliothek in Monorepo ausprobieren

Die Buildings-Blocks von Bibliotheken lassen sich wie alle anderen Building-Blocks auch mit Unit-Tests testen (siehe Kapitel 12). Beim Ausführen der Tests ist der Bibliotheksname anzugeben:

```
ng test logger-lib
```

Daneben können Sie Ihre Bibliothek auch in anderen Subprojekten des Monorepos verwenden. Um im Monorepo einzelnen Subprojekten Zugriff auf die eingerichteten Bibliotheken zu geben, richtet die CLI Path-Mappings ein. Diese finden sich in der `tsconfig.json` im Hauptverzeichnis des Monorepos:

```
"paths": {  
  "logger-lib": [  
    "dist/logger-lib/logger-lib",  
    "dist/logger-lib"  
  ]  
},
```

Standardmäßig verweisen diese Mappings auf den `dist`-Ordner. Das bedeutet, dass Sie die Bibliothek nach jeder Änderung kompilieren müssen. Da das nicht nur lästig, sondern auch fehleranfällig ist, bietet es sich an, diese Mappings auf den Quellcode Ihrer Bibliotheken zeigen zu lassen. Bei dieser Gelegenheit können Sie auch den gegebenenfalls in der `package.json` konfigurierten Scope ins Spiel bringen:

```
"paths": {  
  "@my/logger-lib": [  
    "projects/logger-lib/src/public-api"  
  ]  
},
```

Dieses Path-Mapping verweist nun auf die `public-api` der `logger-lib`. Sämtliche von der `public-api` exportierten Building-Blocks lassen sich somit in den anderen Subprojekten des Monorepos nutzen. Diese verwenden dazu lediglich einen TypeScript-Import, der auf den abgebildeten Namen `@my/logger-lib` verweist:

```
import { LoggerService } from '@my/logger-lib'
```

Lange und unleserliche relative Pfade innerhalb von `import`-Anweisungen gehören somit der Vergangenheit an.



Vermeiden Sie es, das eigene Path-Mapping innerhalb der Bibliothek zu nutzen. Das führt zu zyklischen Verweisen. Dasselbe gilt für Importe, die die eigene `public-api` adressieren.

Nach dem Ändern von Path-Mappings sollten Sie Ihre IDE neu starten. Danach können Sie das Angular-Modul der Bibliothek in das `AppModule` der `playground-app` importieren (siehe Beispiel 19-4).

*Beispiel 19-4: Das AppModule der playground-app importiert das LoggerLibModule der logger-lib.*

```
// projects/playground-app/src/app/app.module.ts
```

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

```

// Hinzufügen:
import { LoggerLibModule } from '@my/logger-lib';

import { AppComponent } from './app.component';

@NgModule({
  imports: [
    BrowserModule,

    // Hinzufügen:
    LoggerLibModule
  ],
  declarations: [
    AppComponent
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Da wir in diesem einfachen Rundgang lediglich einen Service nutzen, der mit einem Tree-Shakable Provider registriert wird, könnten wir auch auf den Import des Logger LibModule verzichten. Sobald wir allerdings klassische Provider, Pipes, Komponenten oder Direktiven aus der Bibliothek nutzen möchten, führt daran kein Weg vorbei.

Zum Ausprobieren lassen wir uns den LoggerService in die AppComponent der *playground-app* injizieren (siehe Beispiel 19-5).

*Beispiel 19-5: Die AppComponent der playground-app verwendet den LoggerService der logger-lib.*

```

// projects/playground-app/src/app/app.component.ts

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

// Hinzufügen:
import { LoggerService } from '@my/logger-lib';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'playground-app';

  // Hinzufügen:
  constructor(private logger: LoggerService) {
    logger.log('Manfred war hier!');
  }
}
```

Der Konstruktor nutzt auch gleich den LoggerService, um eine Testnachricht zu protokollieren.

Führen wir nun die *playground-app* aus (`ng serve playground-app -o`), sollten wir diese Testnachricht in der JavaScript-Konsole sehen (siehe Abbildung 19-3).

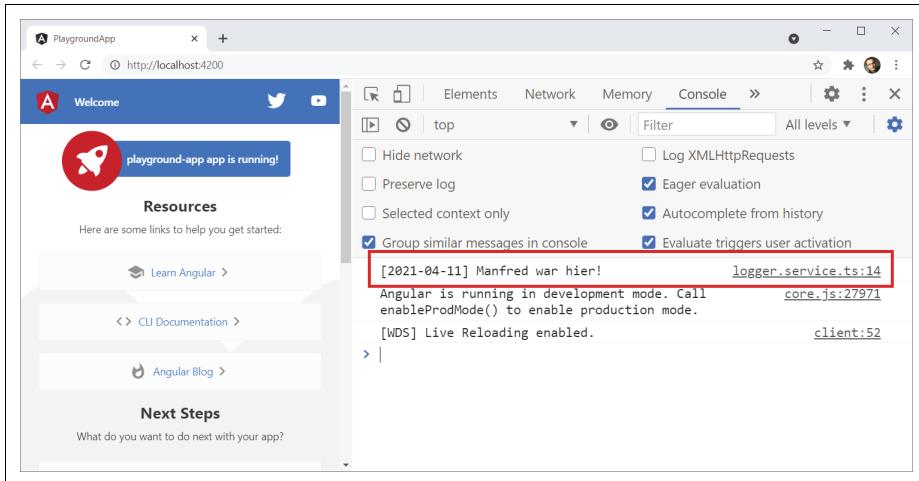


Abbildung 19-3: Die *playground-app* protokolliert mit dem *LoggerService* eine Nachricht.

## npm-Paket bauen und bereitstellen

Um eine Bibliothek als npm-Paket zu veröffentlichen, sollten Sie zunächst sicherstellen, dass ihre *package.json* (z.B. `projects/logger-lib/package.json`) eine eindeutige Versionsnummer aufweist. Ansonsten erhalten Sie beim Bereitstellen einen Fehler.

Danach ist die Bibliothek mit dem Schalter `--prod` zu bauen:

```
ng build logger-lib --prod
```

Der Schalter `--prod` berücksichtigt alle internen Details für die Bereitstellung eines Pakets über eine npm-Registry.

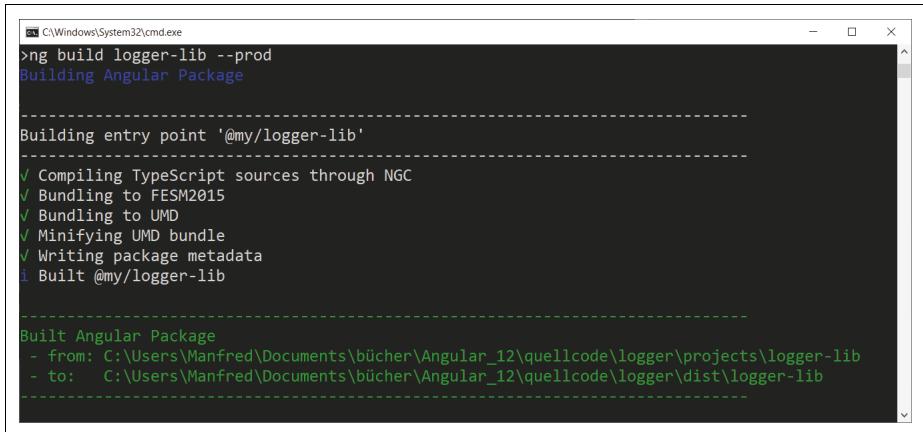
Nach dem Bauen findet sich das npm-Paket im Ordner *dist/logger-lib* (siehe Abbildung 19-4).

Die konkrete Konsolenausgabe kann sich von Version zu Version ändern.

Ein Aufruf von `npm publish` stellt das Paket in der öffentlichen npm-Registry der Allgemeinheit zur Verfügung:

```
npm publish dist/logger-lib
```

Alternativ dazu können Sie auch eine private (unternehmensinterne) npm-Registry verwenden. Falls Sie noch keine haben, empfiehlt sich ein Blick auf Verdaccio (<https://verdaccio.org/>). Dabei handelt es sich um einen freien und sehr leichtgewichtigen npm-Server. Er lässt sich sowohl lokal zum Testen als auch als Windows-Server bzw. Linux-Daemon auf einem Server einrichten.



```
C:\Windows\System32\cmd.exe
>ng build logger-lib --prod
Building Angular Package

-----
Building entry point '@my/logger-lib'

-----
✓ Compiling TypeScript sources through NGC
✓ Bundling to FESM2015
✓ Bundling to UMD
✓ Minifying UMD bundle
✓ Writing package metadata
i Built @my/logger-lib

-----
Built Angular Package
- from: C:\Users\Manfred\Documents\bücher\Angular_12\quellcode\logger\projects\logger-lib
- to:   C:\Users\Manfred\Documents\bücher\Angular_12\quellcode\logger\dist\logger-lib
```

Abbildung 19-4: Die Bibliothek wird als npm-Paket gebaut.

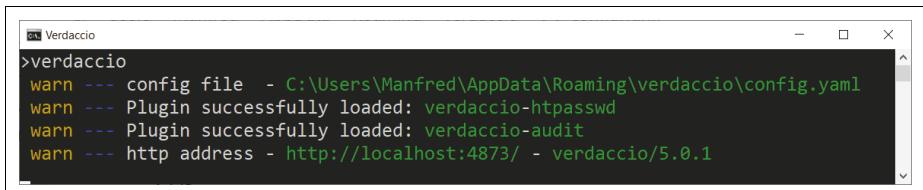
Für die lokale Installation kommt npm zum Einsatz:

```
npm i -g verdaccio
```

Danach lässt sich die Registry bereits starten:

```
verdaccio
```

Nach dem Start gibt Verdaccio die Adresse aus, unter der er erreichbar ist (siehe Abbildung 19-5).



```
C:\ Verdaccio
>verdaccio
warn --- config file - C:\Users\Manfred\AppData\Roaming\verdaccio\config.yaml
warn --- Plugin successfully loaded: verdaccio-htpasswd
warn --- Plugin successfully loaded: verdaccio-audit
warn --- http address - http://localhost:4873/ - verdaccio/5.0.1
```

Abbildung 19-5: Start von Verdaccio

Standardmäßig ist das `http://localhost:4873`. Nach dem ersten Start sollten Sie einen npm-Benutzer anlegen:

```
npm adduser --registry http://localhost:4873
```

Danach können Sie Ihr npm-Paket bei Verdaccio mit `npm publish` bereitstellen:

```
npm publish dist/logger-lib --registry http://localhost:4873
```

Wenn Sie nun Verdaccio im Browser aufrufen, sollten Sie das bereitgestellte Paket sehen (siehe Abbildung 19-6).

Damit Sie nicht immer die Registry der Wahl beim Aufruf von `npm publish` angeben müssen, bietet es sich an, im Projekthauptverzeichnis eine `.npmrc`-Datei anzulegen.

Dabei handelt es sich unter anderem um eine `.ini`-Datei, die einen Eintrag für die Standard-Registry beinhalten kann:

```
registry=http://localhost:4873
```

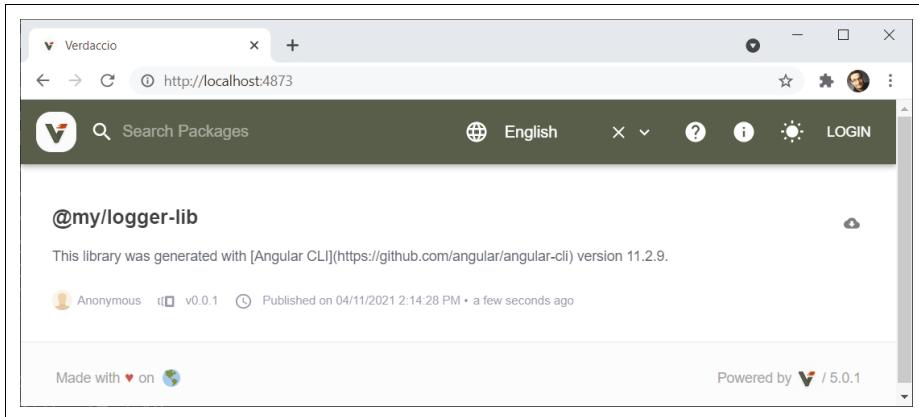


Abbildung 19-6: Verdaccio zeigt die bereitgestellte Bibliothek an.

Es lässt sich sogar eine eigene Registry für einzelne Scopes konfigurieren:

```
@my:registry=http://localhost:4873
```

Der Vorteil dieser Vorgehensweise ist, dass die `.npmrc`-Datei auch in die Quellcodeverwaltung aufgenommen und somit allen Kollegen zur Verfügung gestellt wird.

## npm-Paket konsumieren

Der Konsument installiert das bereitgestellte Paket mit `npm install` in seine Angular-Anwendung. Dort kann er es – wie oben im Rahmen der `playground-app` demonstriert – nutzen:

```
npm install @my/logger-lib --registry http://localhost:4873
```

Falls der Konsument auch die verwendete Registry als Standard-Registry konfiguriert hat, kann hier der Schalter `--registry` weggelassen werden.

Fordern Sie ein Paket an, das Verdaccio nicht kennt, delegiert er standardmäßig an die öffentliche npm-Registry und bezieht das Paket von dort.

## Zusammenfassung

Sämtliche Angular-Projekte lassen sich als Monorepos bzw. Workspaces nutzen. Das bedeutet, dass das Projekt verschiedene zusammengehörige Subprojekte gruppiert. Dabei kann es sich um ausführbare Anwendungen oder um wiederverwendbare Bibliotheken handeln. Letztere lassen sich als npm-Pakete veröffentlichen und in andere Projekte einbinden.

---

# Index

## Symbol

? 53  
... 66  
@angular/localize 373  
@Injectable *siehe* Injectable  
@Input 104  
@ngrx/effects 399  
@ngrx/store 397  
@ngrx/store-devtools 408  
@Output 110  
=> 52  
\$event 110  
\$implicit 446, 451

## A

abstract 59  
AbstractControl 234  
abstrakte Klasse 130  
implementieren 130  
vererben 130  
Access-Modifier 49  
Access-Token 363  
Action (NGRX) 405  
ActivatedRoute 354  
ActiveRouteSnapshot 345  
afterAll 294  
AfterContentChecked 432  
AfterContentInit 432  
afterEach 294  
AfterViewChecked 432  
AfterViewInit 431, 432  
Angular CLI 23  
Anwendung bauen 26  
Anwendung starten 24  
Projekt generieren 23

Angular Language Service 22  
Angular Schematics 22, 74  
Angular-Modul 29  
angular-oauth2-oidc 366  
Anwendung (CLI) 471  
any 40, 43  
Api-Modul 329  
Array 43  
Arrow-Funktion *siehe* Lambda-Ausdruck  
async (Pipe) 285  
async (Schlüsselwort) 71  
asynchron 68  
AsyncValidator 225  
AsyncValidatorFn 244  
Attributdirektive 438  
Audience (OAuth 2) 364  
Auth 2 361  
Authentifizierung 359  
Authorization Code Flow 365  
Authorization Server 362  
Auto-Import 47  
Autorisierung 359  
Aux-Routes 191  
await 71

## B

Basisklasse 60  
beforeAll 294  
beforeEach 294  
BehaviorSubject 261  
Bibliothek 471  
Bibliothek installieren 32  
boolean 42  
Bootstrap 107  
browserslistrc 31

## C

Callback 69  
CanActivate 344  
canActivate 345  
CanActivateChild 344  
CanDeactivate 344, 346  
canDeactivate 347  
CanLoad 344  
catchError 283  
Checkbox 229  
Client (OAuth 2) 361  
Cold Observable 281  
combineLatest 272, 275  
CommonModule 165  
compileComponents 298  
complete (RxJS) 257  
Component 74  
componentOutlet 458, 462  
concatMap 271  
const 40  
constructor 49  
Container 443  
Content 424  
Content Projection 418, 424  
ContentChild 425, 426  
context (Direktive) 451  
ControlValueAccessor 464  
Cookie 359  
create-application (Angular CLI) 471  
createComponent 463  
createEmbeddedView 453  
createSelector 411  
createUrlTree 345  
creators 401  
Cross Site Request Forgery 360  
CSRF *siehe* Cross Site Request Forgery

## D

Date 46  
Datenbindung 95  
debounceTime 265, 270, 275  
Debug-Element 311  
Debugger 91  
Debugger for Chrome 22  
Debug-Modus 116  
declarations 166  
Deep Linking 173  
DefaultFlightService 307  
delay 267  
dependencies 475

Dependency Injection 79, 123  
Konstante 145  
Testing 299  
Token 128  
deps 141  
describe 294  
Direktive 84, 438  
testen 312  
Direktive (strukturell) 443  
dirty 208  
Dirty-Checking 315  
distinctUntilChanged 265  
DOM 433  
Drop-down-Feld 230

## E

Effect 406  
Effects 400  
EffectsModule 401  
ElementRef 433  
email (Validierung) 207  
End-2-End-Tests 301  
environment 31  
error (RxJS) 257  
errors 210  
Event-Binding 86, 97, 108  
EventEmitter 110  
Exception 64  
exhaustMap 271  
expect 294  
expectOne 307  
export 45  
exportAs 442  
exports 167  
extends 130

## F

Factory 140  
fakeAsync 310  
fdescribe 295  
Feature-Module 163, 167  
Feature-Selektor 412  
Fehlerbehandlung 64  
filter 265  
firstValueFrom 263  
Flattening 270, 271  
Flow (OAuth 2, OIDC) 365  
FormArray 234, 250  
Format (I18N) 383  
FormBuilder 236

FormControl 234, 236  
formControlName 238  
FormGroup 234, 238  
  verschachteln 248  
FormsModule 206  
Formular  
  dynamisch 253  
  reakтив 233  
  Template-getrieben 205  
for-of 43  
forRoot 336, 400  
from (RxJS) 263, 264  
FromControl 269  
function 44, 50  
Funktion 50  
  anonym 51

## G

Generics 62  
Geschlecht (Compiler-I18N) 382  
Getter 61  
Grahsl 19  
Gray-Box-Test 304  
Guard 343, 344

## H

hasError 210  
hash 198  
Hash-Fragment 196  
HashLocationStrategy 201, 203  
Hash-Routing 201  
Headless Chrome 296  
HostBinding 440  
HostListener 440, 442  
Hot Observable 281  
HttpClient 77, 355  
  Testing 306  
HttpClientTestingModule 306  
HttpInterceptor 355  
HTTP-Kopfzeile 80  
HttpParams 80

## I

i18n (angular.json) 377  
i18n (Attribut) 373  
I18N *siehe* Internationalisierung  
i18nPlural (Pipe) 391  
i18nSelect (Pipe) 391  
ID-Token 363  
immutable 317

implements 54, 130  
Implicit Flow 365  
import 46, 325  
imports 166  
initialState 403  
Inject 143  
Injectable 125  
  deps 141  
  providedIn 125  
  useClass 130, 138  
  useExisting 141  
  useFactory 140  
  useValue 138  
InjectionToken 145  
Input 440  
instanceof 58  
intercept 356  
Interface 44, 53  
Internationalisierung 371  
Internet Explorer 11 31  
interval 264  
isNaN 41  
ISO-Datum 46  
isolate (ngx-translate) 394  
it 294

## J

Jasmine 293  
  asynchron 308  
  done (Funktion) 308  
  Spy 304  
JSON 46, 81  
JSON Web Token 364  
JWT *siehe* JSON Web Token

## K

Karma 293, 295  
  Build-Server 296  
Klasse 47  
  abstrakt 59  
klonen 66  
Komponente 74  
Konstruktor 49  
Kopfzeile 80

## L

L10N *siehe* Lokalisierung  
Lambda-Ausdruck 52  
lastValueFrom 263  
Lazy Loading 325

Lazy Loading (ngx-translate) 393  
let 39  
Life-Cycle-Hook 112, 431  
loadChildren 325  
Loader (ngx-translate) 386  
loadTranslations 380  
Locale 371  
Locale (angular.json) 377  
Lokalisierung 371

## M

Marble-Diagramme 257  
match 307  
Matcher 294  
Matrixparameter 196  
max 207  
maxLength 207  
merge 266, 277  
mergeMap 271  
Meta-Reducer 412  
Microsyntax 445  
min 207  
minLength 207  
Mocks 299  
Modul 29, 162  
ModuleWithProviders 336  
Modulstruktur 162  
Monorepo 471  
multi 142, 147  
Multicasting 267, 279

## N

NaN 41  
NavigationCancel 349  
NavigationEnd 349  
NavigationError 349  
NavigationStart 349  
next (RxJS) 257  
ng extract-i18n 375  
ngAfterContentChecked 432  
ngAfterContentInit 421, 432  
ngAfterViewChecked 432  
ngAfterViewInit 432  
NG\_ASYNC\_VALIDATORS 225  
ngClass 85  
ngComponentOutlet 454  
ng-content 419  
NgControl 469  
ng-dirty 212  
ngFor 84

NgForm 208  
ngIf 112  
ng-invalid 212  
ngModel 83, 100, 206  
ngModelChange 100  
ngModelOptions 211  
ngOnChanges 112, 117  
ngOnDestroy 112, 137, 159  
ngOnInit 76, 112  
ng-pending 212  
ng-pristine 212  
NGRX 400  
ngTemplateOutlet 452  
ng-valid 212  
NG\_VALIDATORS 217  
ngx-translate 385  
Node.js 23  
npm publish 478  
npm-Paket 471  
npmrc 479  
number 41

## O

OAuth 2 361  
OAuth 2.1 365  
Observable 255  
Datenbindung 319  
Observer 255  
of (RxJS) 264  
OIDC *siehe* OpenID Connect  
OnChanges 112  
OnDestroy 112  
OnInit 76, 104, 112  
OnPush 315  
OpenID Connect 361, 363  
Output 440

## P

package.json 475  
params 184  
parseFloat 41  
parseInt 41  
patchValue 239  
path 177  
PathLocationStrategy 201  
Path-Mapping 476  
pathMatch 177  
pattern 207  
peerDependencies 475  
Pipe 86, 149

Built-in 149  
deklarieren 154  
generieren 151  
ngOnDestroy 159  
pure 151  
registrieren 154  
testen 312  
pipe (RxJS) 257  
PipeTransform 153  
PKCE *siehe* Proof Key for Code Exchange  
Plural (Compiler-I18N) 382  
Polymorphismus 55  
preload 339  
PreloadAllModules 338  
Preloading 337  
PreloadingStrategy 337, 339  
preserveHash 198  
private 49  
Promise 70, 262  
Proof Key for Code Exchange 365  
Property-Binding 84, 96, 103  
protected 49  
Protractor 301  
providedIn 125, 328  
Provider 125  
klassisch 133  
Konstante 145  
multi 142, 147  
Tree-Shakable Provider 125  
useClass 130, 138  
useExisting 141  
useFactory 140  
useValue 138  
public 49  
public-api 476  
pur 151  
Pyramide of Doom 69

## **Q**

QueryList 425  
queryParams 198  
queryParamsHandling 198  
Query-String 196

## **R**

RadioButton 229  
ReactiveFormsModule 233  
readonly 49  
redirectTo 177  
Reducer 405

Redux 397  
Redux DevTools 408  
registerOnChange 464  
registerOnTouched 464  
ReplaySubject 261  
required 207  
requireTrue 207  
resolve 353  
Resolver 351  
Resource Server 362  
root *siehe* Root-Scope  
Root-Module 163, 169  
Root-Scope 125, 328  
Router 173  
Aux-Routes 191  
Child-Routes 186  
data 354  
Event 349  
Feature-Modules 178  
forChild 178  
forRoot 178  
Hash-Fragment 196  
Hash-Routing 201  
hierarchisch 186  
Konfiguration 176  
Lazy Loading 325  
loadChildren 325  
Parameter 182  
params 184  
path 177  
pathMatch 177  
queryParams 198  
Query-String 196  
redirectTo 177  
Resolver 351  
routerLink 180  
routerLink 180, 185  
RouterModule 178  
router-outlet 179  
RoutesRecognized 349

## **S**

scope (npm) 475  
Selektor 410  
Service 123  
austauschen 128  
generieren 123  
Inject 143  
lokal 135  
multi 142, 147

ngOnDestroy 137  
Singleton 127  
Token 128  
useClass 130, 138  
useExisting 141  
useFactory 140  
useValue 138  
Setter 61  
share 281  
Shared Module 163  
shareReplay 281  
Singleton 127  
sourceLocale (angular.json) 377  
Spread-Operator 66  
Spy 304  
startWith 274  
StoreModule 400  
Strict Mode 66  
Strikte Null-Prüfungen *siehe* Strict Mode  
string 42  
Subject 261  
subscribe 255  
Subscription 260  
super 60  
switchMap 270, 271

**T**

take 265, 285  
takeUntil 285  
tap 267  
Teardown-Funktion 258  
Template 87, 99, 443  
TemplateRef 444, 448, 451  
Template-Variable 426, 434  
Test  
    asynchron 308  
    DOM 310  
Testabdeckung 313  
 TestBed 297  
 Testfall 293  
 Testing Module 297  
 Testsuite 293  
 this 52  
 throw 65  
 throw Error 264  
 timer 274  
 Token 128  
     Konstante 145  
 Token (Security) 361

toPromise 263  
translate (Pipe) 389  
TranslateService 389  
Tree-Shakable Provider 125  
Two-Way-Binding 82, 99, 111

## U

Unit-Test 301  
unknown 43  
unsubscribe 260, 285  
URL-Parameter 80  
URL-Segment 185, 196  
UrlTree 345  
useClass 130, 138  
useExisting 141  
useFactory 140  
useValue 138

## V

valid 208  
validate 225  
Validator 216  
ValidatorFn 240  
Validators 235  
Validierungsdirektive 213  
valueChanges 269  
var 40  
Verdaccio 478  
Vererbung 56  
View 424  
ViewChild 430  
    read 433  
    static 432  
ViewChildren 431  
ViewContainerRef 433, 444, 452  
Visual Studio Code 21

## W

waitForAsync 309  
withLatestFrom 275  
Workspace (CLI) 471  
writeValue (ControlValueAccessor) 464

## X

XSRF *siehe* Cross Site Request Forgery

## Z

Zustandsbaum (NGRX) 399  
Zyklus 95, 115, 328

# Über den Autor



*Manfred Steyer bei einem Angular-Training*

**Manfred Steyer** (GDE) ist Trainer und Berater mit Schwerpunkt Angular. Er betreut Firmen im gesamten deutschen Sprachraum. Gemeinsam mit seinem Team bietet er tiefgehende Angular-Workshops und Beratung via <http://www.angulararchitects.io> an.

Manfred hat Bücher bei O'Reilly, Microsoft Press und Hanser veröffentlicht. Er schreibt für Heise Online, windows.developer und das Java Magazin und gibt sein Wissen regelmäßig auf Konferenzen weiter.

Manfred hat berufsbegleitend jeweils neben einer Vollanstellung IT und IT-Marketing in Graz sowie Computer Science in Hagen studiert und beide Vorhaben erfolgreich abgeschlossen.

Früher war Manfred mehrere Jahre zunächst als Teamleiter im Bereich der Softwareentwicklung tätig. Danach hat er sich einige Zeit als FH-Professor um die Koordination und Durchführung von Lehrveranstaltungen im Umfeld der Softwareentwicklung gekümmert und parallel dazu Firmentrainings und Beratungsworkshops veranstaltet.

Für seine Aktivitäten wurde er von Google als Developer Expert (GDE) und von Microsoft mit dem MVP-Award ausgezeichnet. Außerdem ist Manfred Trusted Collaborator im Angular-Team.

Sein Blog finden Sie unter <http://www.angulararchitects.io/blog>.

# Kolophon

Das Tier auf dem Cover von »Angular« ist ein Ai oder Weißkehl-Faultier (*Bradypus tridactylus*), das zur Familie der Dreizehen-Faultiere gehört. Die meiste Zeit des Tages – bis zu zwanzig Stunden – verbringen die zu den Säugetieren gehörenden Faultiere schlafend in den Baumkronen des tropischen Regenwalds. Dabei suchen sie sich Astverzweigungen und schmiegen sich mit dem Körper eng an den Baumstamm an, den Kopf auf die Brust geneigt. Nur wenige Stunden werden sie wach und klettern in ungeheurer Langsamkeit von Ast zu Ast auf der Suche nach Blättern, die sie entweder mit dem Maul abrufen oder mit ihren langen gebogenen Krallen aufspießen. Auf dem Waldboden trifft man sie nur an, wenn sie von einem Baum zum nächsten wechseln.

Faultiere haben einen affenartigen runden, gedrungenen Körper mit langen Armen, einem langen Hals und einem kleinen runden Kopf. Das Gesicht ziert eine gelbe Zeichnung mit schwarzen Augenstreifen. Grau-braunes Fell bedeckt den gesamten Körper, das aus einem dichten Unterfell und langen struppigen Oberhaaren besteht. Der Scheitel der Haare sitzt auf dem Bauch, um das Regenwasser besser abzuleiten. In den teilweise gebrochenen Oberhaaren siedeln sich Grünalgen an. Bei bestimmtem Lichteinfall schimmert das Fell der Faultiere dadurch beinahe grünlich, was ihnen als Tarnung dient. Außerdem leben Motten, Käfer und Zecken in den dichten Haarbüscheln der Tiere und gehen eine regelrechte Symbiose miteinander ein: Die Motten geben Stickstoff an die Algen ab, die wiederum als Zusatznahrung von den Faultieren aufgenommen werden.

Faultiere leben einzeln und kommen nur zur Paarung zusammen. Das Weibchen bringt nach einem halben Jahr ein Junges zur Welt, das sie fünf Monate lang auf dem Rücken oder Bauch mit sich herumträgt. Natürliche Feinde sind der Jaguar, Anakondas und Greifvögel wie die Harpyie.

Faultiere leben im Nordosten Südamerikas von Venezuela bis Brasilien, im sogenannten Hochland von Guayana. Die dortigen Populationen sind stabil, und die Tiere werden in ihrem Bestand als nicht gefährdet eingestuft. Die größte Gefahr geht von menschlichen Besiedlungen aus. Durch Autounfälle kommen viele Tiere ums Leben.