Angular

Praktische Einführung

Manfred Steyer

Einführung in Angular

Manfred Steyer

Dieses Buch wird verkauft unter http://leanpub.com/einfhrung-in-angular

Diese Version wurde veröffentlicht am 2022-01-02

Leanpub

Dies ist ein Leanpub-Buch. Leanpub bietet Autoren und Verlagen, mit Hilfe von Lean-Publishing, neue Möglichkeiten des Publizierens. Lean Publishing bedeutet die wiederholte Veröffentlichung neuer Beta-Versionen eines eBooks unter der Zuhilfenahme schlanker Werkzeuge. Das Feedback der Erstleser hilft dem Autor bei der Finalisierung und der anschließenden Vermarktung des Buches. Lean Publishing unterstützt den Autor darin ein Buch zu schreiben, das auch gelesen wird.

Inhaltsverzeichnis

Einleitung

In den letzten Jahren habe ich zahlreiche Unternehmen mit der Umsetzung von Unternehmens- und Industrieanwendungen mit Angular geholfen. Sowohl mit der Einführung als auch mit weiterführenden Konzepten. Mit diesem Buch möchte ich auch Ihnen zeigen, wie Sie Angular für Ihre Projekte nutzen können.

Dazu erstellen wir gemeinsam im Laufe der Kapitel eine vollständige Angular-Anwendung und verfeinern sie nach und nach. Der Fokus liegt sowohl auf der konkreten Umsetzung als auch auf der Schaffung eines guten Verständnisses für die dahinterliegende Konzepte. Alle Aspekte, die Sie für eine erste Angular-Anwendung benötigen werden dabei besprochen.

Quellcode

Den Quellcode der Beispielanwendung, auf die sich die Beispiele in diesem Buch beziehen, finden Sie in unserem GitHub-Account[1].

Kontakt

Wenn Sie Fragen oder Feedback haben, erreichen Sie uns am besten via manfred.steyer@angulararchitects.io[2].

Außerdem finden Sie mich auch auf Twitter[3] und Facebook[4]. Lassen Sie uns in Kontakt bleiben, um aktuelle Updates rund um Angular zu erhalten.

Trainings and Consulting

Wenn Sie und Ihr Team Unterstützung oder Schulungen in Bezug auf Angular benötigen, helfen wir Ihnen gerne mit unseren Workshops und Beratungen – sowohl **vor Ort** als auch per **Remote**. Wir bieten unter anderem Workshops für folgende Themen an:

- Angular Workshop: Strukturierte Einführung (3 Tage)
- Advanced Angular: Enterprise Solutions and Architecture (3 Tage)
- Angular Architecture Consulting

[1] https://github.com/manfredsteyer/angular-intro
[2] mailto:manfred.steyer@angulararchitects.io
[3] https://twitter.com/manfredsteyer
[4] https://www.facebook.com/manfred.steyer

- Professional Angular Testing Workshop mit Cypress, Jest, etc. (3 Tage)
- Angular: Reactive Architekturen mit RxJS and NGRX (2 Tage)
- Angular Review Consulting Workshop
- Angular Upgrade Consulting Workshop

Sie finden hier unser volles Workshop-Angebot[5].

Advanced Angular Workshop

Wir bieten unsere Workshop in verschiedenen Formen an: **Online**, **Öffentlich** oder als **Unternehmens-Workshop** sowohl in **Englisch** als auch in **Deutsch**.

Wenn Sie Fragen haben, können Sie gerne auf uns zukommen: office@softwararchitekt.at[6].

[5] https://www.angulararchitects.io/en/angular-workshops/
[6] mailto:office@softwararchitekt.at

Erste Schritte mit Angular

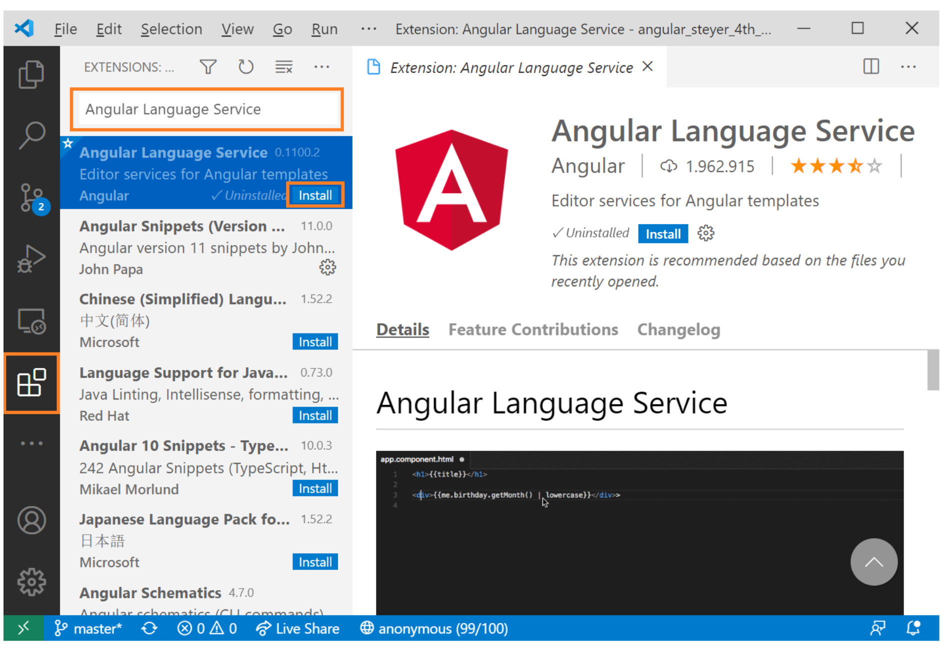
Bevor es losgeht: Werkzeuge installieren

Bevor wir mit Ihrer ersten Angular-Anwendung loslegen können, müssen wir erst mal ein paar Werkzeuge einrichten.

Visual Studio Code

Wir nutzen in diesem Buch die freie Entwicklungsumgebung Visual Studio Code[7]. Sie funktioniert auf allen wichtigen Betriebssystemen (Linux, OSX, Windows) und ist äußerst leichtgewichtig. Visual Studio Code unterstützt ab Werk die Sprache TypeScript.

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:



Erweiterungen in Visual Studio Code installieren

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

[7] https://code.visualstudio.com

- **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.

  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.

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[8], 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.

Die CLI lässt sich leicht über den Package-Manager `npm` beziehen, der sich im Lieferumfang von Node.js[9] befindet. Außerdem nutzt die CLI Node.js als Laufzeitumgebung. Wir 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.

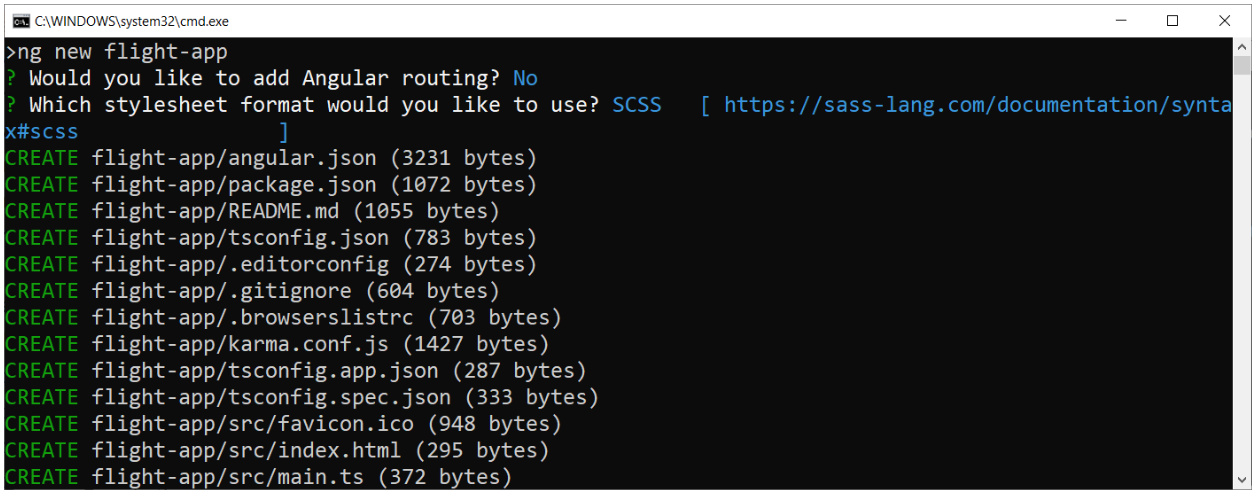
Eine neue Angular-Application erzeugen

Ein Aufruf von

[8] https://cli.angular.io
[9] https://nodejs.org

```
ng new flight-app
```

generiert das Grundgerüst einer neuen Angular-Anwendung, die den Namen `flight-app` erhält. Dazu stellt uns die CLI ein paar Fragen:



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 kümmern wir uns in einem späteren Kapitel.
- **Stylesheet Format**: Wir empfehlen hier SCSS, eine Übermenge 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.

Ihre Angular-Anwendung starten

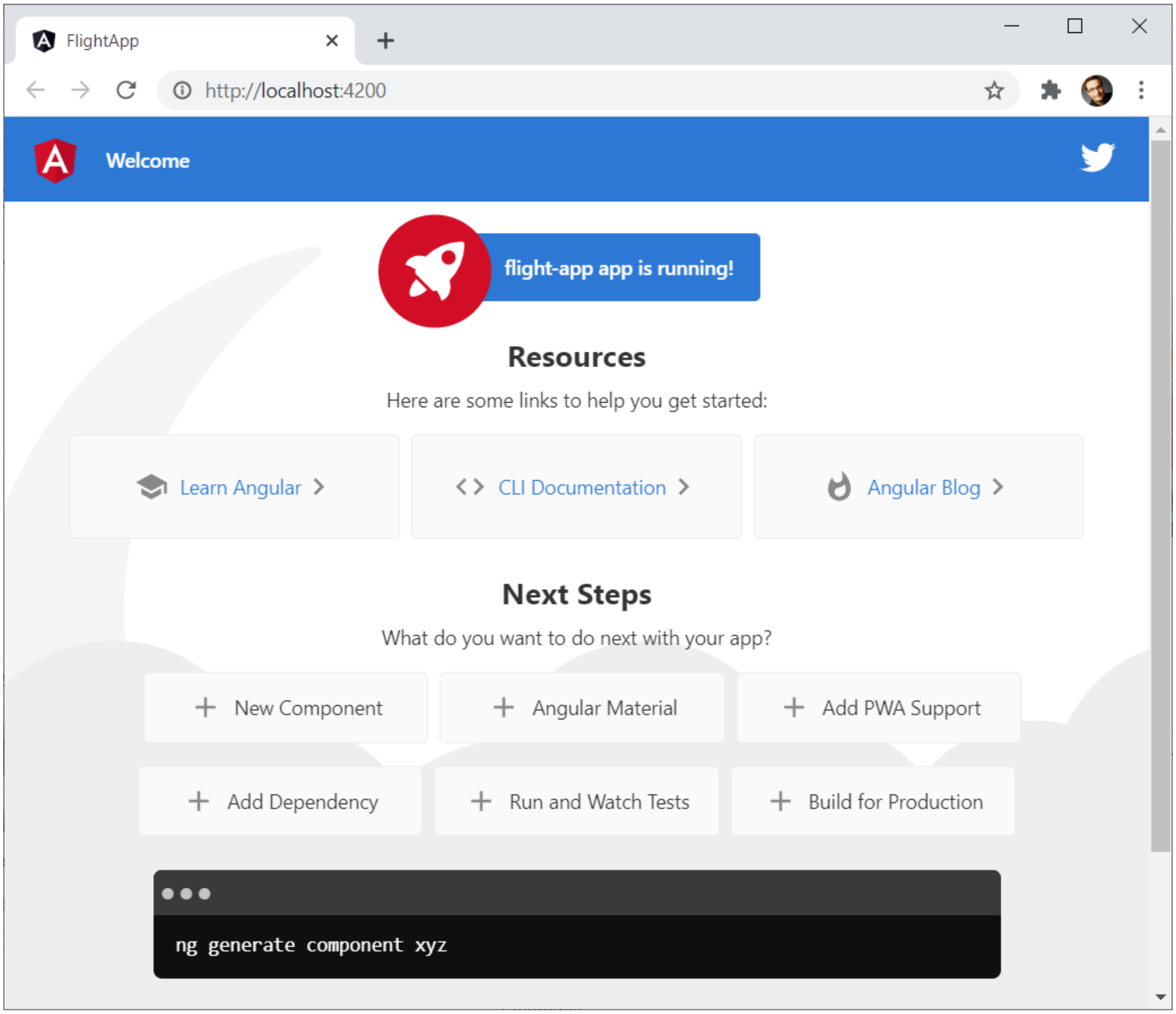
Um Ihre Anwendung zu starten, wechseln Sie in den generierten Projektordner. Dabei handelt es sich um jenen Ordner, der auch die Datei `angular.json` enthält. Ein Aufruf von `ng serve` startet die Anwendung in einem Demo-Webserver:

```
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 folgt aus:

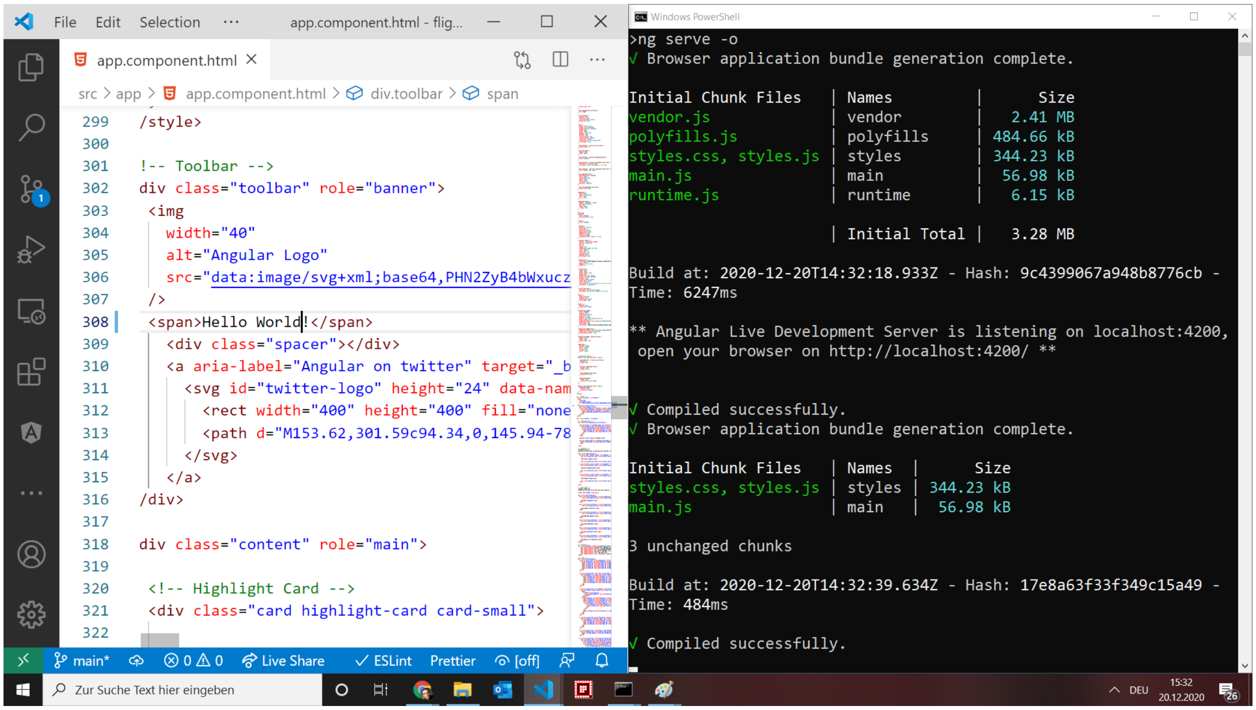


Generierte Angular-Anwendung

Auch hier kann es von Version zu Version zu Abweichungen kommen.

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 z. B. das erste Vorkommen von Welcome durch `Hello World!` ersetzen. Daraufhin sollte `ng serve` den betroffenen Teil der Anwendung neu kompilieren, bundeln und den Browser aktualisieren:



Generierte Angular-Anwendung ändern

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 Produktionseinsatz. 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 kürzere Gegenstücke 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/flight-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.

Das generierte Projekt erkunden

Lassen Sie uns nun ein paar der generierten Programmdateien unter `src/app` etwas genauer betrachten. Starten wir dabei mit der generierten `AppComponent`. Es handelt sich dabei um jene Komponente, die Angular beim Programmstart anzeigt. 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.

Bei der `app.component.ts` handelt es sich um eine einfache Klasse mit einer Eigenschaft `title`:

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

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

Der `title` ist vom Typ `string`. Letzteres muss hier gar nicht explizit angeben 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. Der `Component` teilt beispielsweise Angular mit, dass diese Klasse eine Komponente repräsentiert. Das Programmcode importiert den Dekorator in der ersten Zeile aus dem Paket `@angular/core`.

Die Metadaten im Dekorator 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 (falls noch nicht geschehen: `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 Datenbindungen 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:

```
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 diese sich 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 XY geht 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. Diese Komponente ist in der `index.html` aufzurufen:

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

Sowohl `ng serve` als auch `ng build` ergänzen diese `index.html` auch um Verweise auf die erzeugten JavaScript-Bundles, die unseren Quellcode enthalten.

Programmieren mit “Stil”: Bootstrap installieren

Da auch “das Auge mitprogrammiert”, wollen wir an dieser Stelle ein paar vordefinierte Styles ins Spiel bringen. Wir nutzen hier die populäre Stylesheet-Bibliothek `Bootstrap`.

> 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.

Da das Standard-Design von Bootstrap ein wenig langweilig ist, nutzen wir auch ein freies Bootstrap Theme. Es nennt sich Paper-Design und kommt von Creative Tim[10]. Dazu könnten Sie nun natürlich Bootstrap via npm installieren und die CSS-Dateien des Themes in Ihr Projekt kopieren.

Um diese Aufgabe ein wenig zu vereinfachen, haben wir ein Meta-Paket bereitgestellt. Sie können es einfach via `ng add` installieren:

[10] https://www.creative-tim.com

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

Bei `ng add` handelt es sich um einen Mechanismus der CLI, der beim Hinzufügen von Paketen hilft. Er installiert ein Paket und führt ein Skript aus, das das Paket einrichtet. Natürlich könnte man die dazu notwenigen Schritte auch manuell ausführen.

Die Ausführung von `ng add` gestaltet sich wie folgt:

```
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)

>
```

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` und eine `SideBarComponent` für die Navigation.

Danach erweitert dieser Aufruf von `ng add` die Dateien `angular.json` und `index.html`. Erstere erhält Verweise auf die Style-Dateien von Bootstrap und dem freien Paper Design-Theming von Creative Tim:

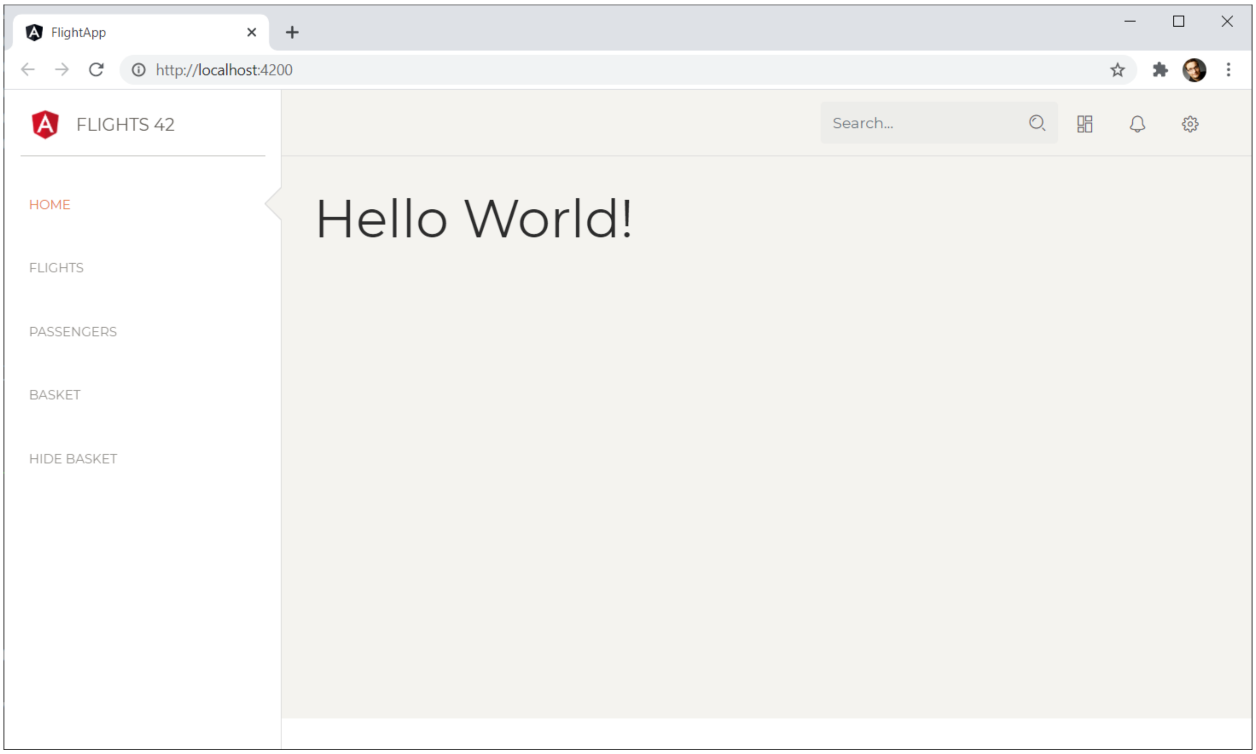
```
"styles": [
        "node_modules/@angular-architects/paper-design/assets/css/bootstrap.css",
        "node_modules/@angular-architects/paper-design/assets/scss/paper-dashboard.scss",
        "src/styles.scss"
],
```

In diesem Listing sieht man übrigens auch die von `ng new` generiere Datei `src/styles.scss`, in der Sie Ihre eigenen globalen Styles hinterlegen können.

Die `index.html` erhält zwei `link`-Elemente zum Laden des vom Theming verwendeten Webfonts.

> 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 folgende Bild:



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.

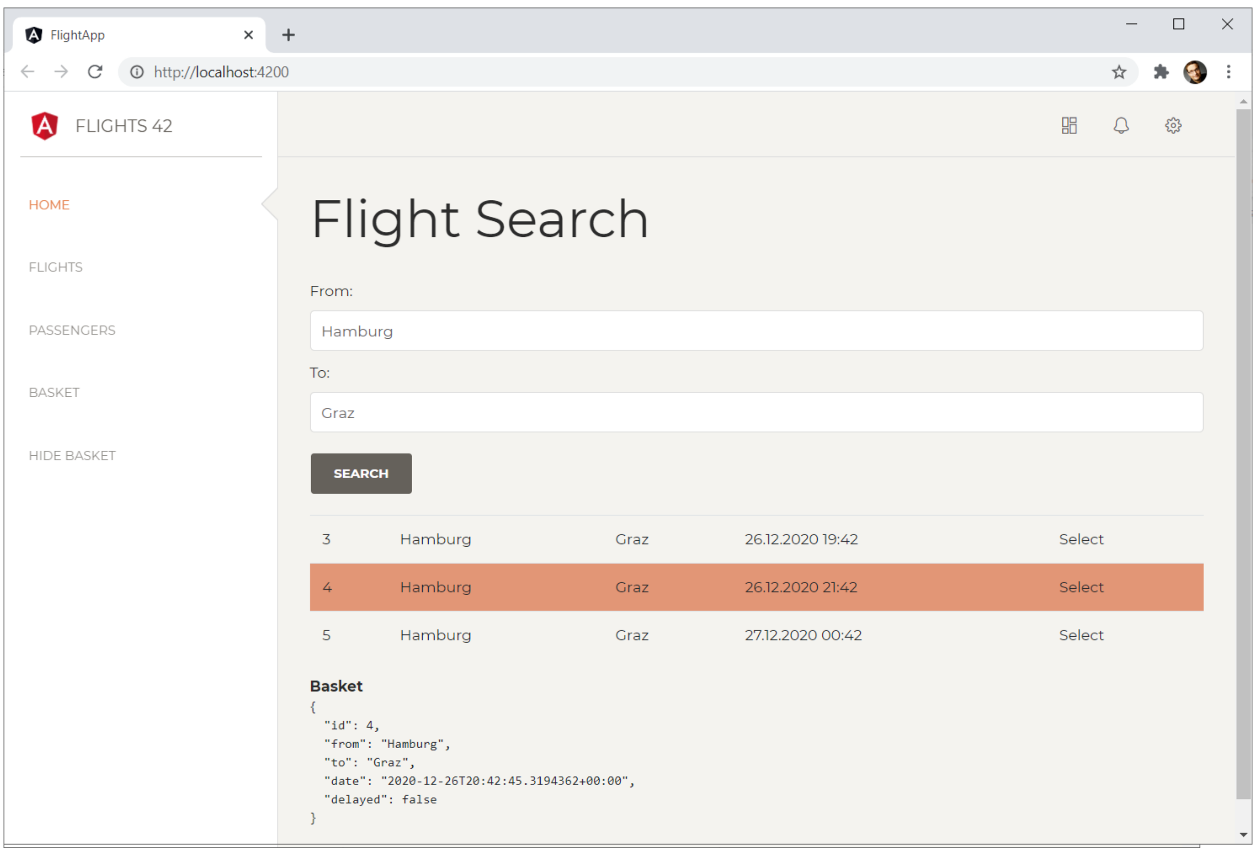
> Tipp: Werfen Sie einen Blick auf den Quellcode der beiden generierten Komponenten und der in der `angular.json` erzeugten Einträge. Wie schon erwähnt, könnten Sie diese Dateien auch manuell einrichten und erweitern. Da das jedoch in der Regel monoton und fehleranfällig ist, freuen wir uns über `ng add`.

Zusammenfassung

Die Angular CLI hilft beim Einrichten, Ausführen und Bauen von Angular-Projekten. 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.

Ihr erste Angular-Anwendung: Komponenten, Datenbindung und HTTP-Zugriff

Um Ihnen die einzelnen Aspekte von Angular zu vermitteln, verwenden wir in diesem Buch ein durchgängiges Beispiel. Sie können es in unserem GitHub-Account[11] finden. Dabei handelt es sich um eine Anwendung zum Buchen von Flügen. Wir setzen dazu auf die im letzten Kapitel generierte Anwendung auf:



Anwendung zum Suchen nach Flügen

[11] https://github.com/manfredsteyer/angular-intro

Interface für Datenobjekt erzeugen

Da wir mit Flügen arbeiten wollen, brauchen wir einen Datentyp der die Struktur der Flug-Objekte widerspiegelt. Hierzu legen wir zunächst im Ordner `src/app` eine Datei `flight.ts` mit dem folgenden Interface an:

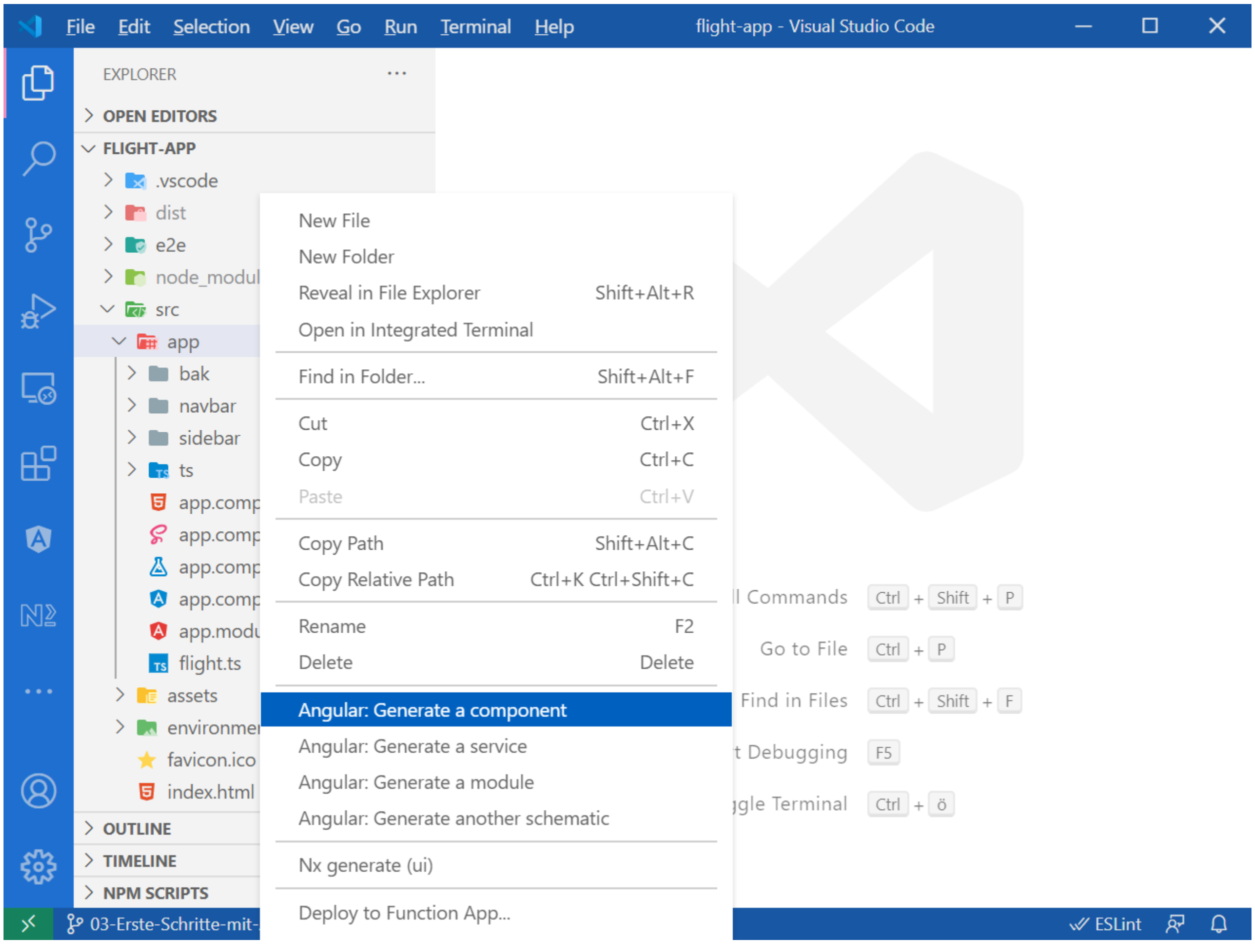
```
// src/app/flight.ts
export interface Flight {
    id: number;
    from: string;
    to: string;
    date: string;
    delayed?: boolean;
}
```

Angular-Komponente erzeugen

Nun erstellen wir eine Angular-Komponente für den besprochenen Anwendungsfall erstellen. Wechseln Sie dazu auf die Konsole. Führen Sie im Hauptverzeichnis der Anwendung (Verzeichnis mit der `angular.json`) den folgenden Befehl aus:

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

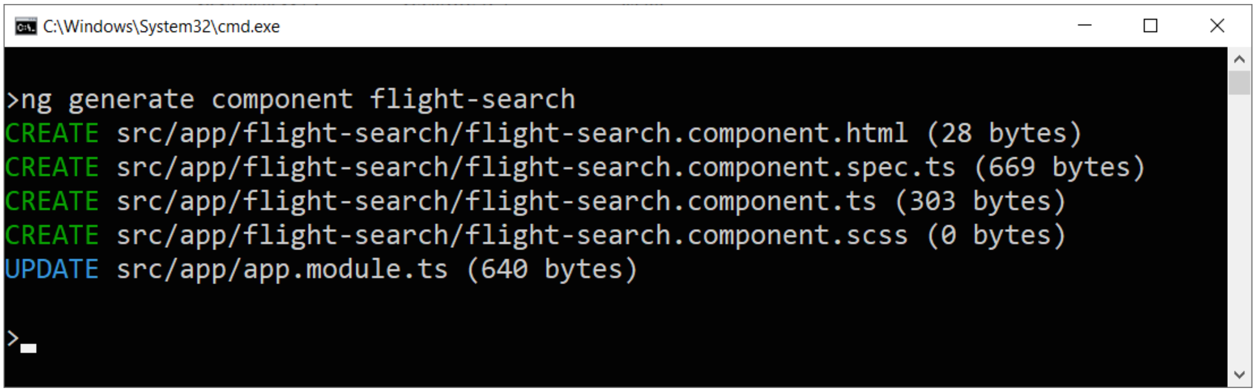
> Die Befehle der CLI lassen sich abkürzen, die betrachtete Anweisung könnte man beispielsweise auch wie folgt formulieren:
>
> ```
> ng g c flight-search
> ```
>
> Mit dem in Kapitel 1 erwähnten Visual Studio Plug-in *Angular Schematics* Angular Schematics lässt sich dieser CLI-Befehl auch direkt über Visual Studio Code anstoßen. Wählen Sie dazu die Anweisung *Angular: Generate a component* aus dem Kontextmenü des gewünschten Ordners.



Komponente in Visual Studio Code generieren

> 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:



Komponente zum Suchen nach Flügen mit der CLI generieren

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.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:

```
// 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.

Lassen Sie uns nun dieses Grundgerüst ein wenig ausbauen, um eine Suche nach Flügen zu ermöglichen:

```
// 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 zu erzeugten Interface `Flight` typisiert.

Die Eigenschaft `selectedFlight` repräsentiert den ausgewählten Flug. Damit sie initial den Wert null bekommen kann, ist sie vom Typ `Flight | null`.

> Angular verwendet standardmäßig TypeScript im Strict Mode. Das bedeutet unter anderem, dass Sie explizit angeben müssen, ob Eigenschaften den Wert `null` bzw. `undefined` aufnehmen dürfen. In diesen Fällen zwingt Sie TypeScript auch dazu, vor der Verwendung gegen diese Werte zu prüfen.

Die Methode `search` kümmert sich um das Abrufen der Flüge. Wir werden uns um ihre Implementierung gleich kümmern. Die Methode `select` notiert sich den vom Benutzer ausgewählten Flug.

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 auch von einem Service die Rede.

Um Zugriff auf den Service zu bekommen, müssen Sie zunächst das `HttpClientModule` in Ihr `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:

```
// 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* 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 Produktionsbetrieb den "richtigen" `HttpClient` erzeugt, könnte es für automatisierte Tests eine Dummy-Implementierung verwenden, die HTTP-Zugriffe lediglich simuliert.

Da wir nun unsere `HttpClient`-Instanz haben, können wir damit innerhalb von `search` auf die Web-API zugreifen:

```
// 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.ANGULARarchitects.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 bei einer von uns bereitgestellten Web API ("Rest API") Flüge ab und hinterlegt sie in der Eigenschaft `flights`:

- Die zu nutzenden HTTP-Kopfzeilen Kopfzeile HTTP-Kopfzeile 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 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
- 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*.
- Neben der hier verwendeten Methode `get` bietet der `HttpClient` noch weitere Methoden für andere Arten von HTTP-Zugriffen.

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

Zur Veranschaulichung erzeugt die folgende Methode einen neuen Flug.

```
createDemoFlight(): void {
    const url = 'http://demo.ANGULARarchitects.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 sogenannteDatenbindungsausdrücke 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.

Hier wollen wir nun auf weitere Arten der Datenbindung eingehen.

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. Angular nutzt diesen Wert zum Aufbau interner Datenstrukturen.

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:

```
// 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 im nächsten Kapitel.

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:

```html
<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.

```html
<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:

```scss
.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.

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 Entwicklern hilft, ist die Pipe `json`. Sie wandelt das gesamte Objekt in seine JSON-Repräsentation um. Somit können Entwickler Objekte zum Testen ausgeben, ohne dafür eine Komponente oder Markup schreiben zu müssen:

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

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 den typischen Mauscursor für klickbare Links (Zeigefingersymbol) anzeigt: `a { cursor: pointer; }`

Das gesamte Template Template

Der Vollständigkeit halber platzieren wir hier nochmal das gesamte Template für die `FlightSearchComponent`, das wir in den vorangegangenen Abschnitten besprochen haben:

```html
<!-- 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>
```

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.

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:

```
// 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:

```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">

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

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

        </div>
    </div>

</div>
```

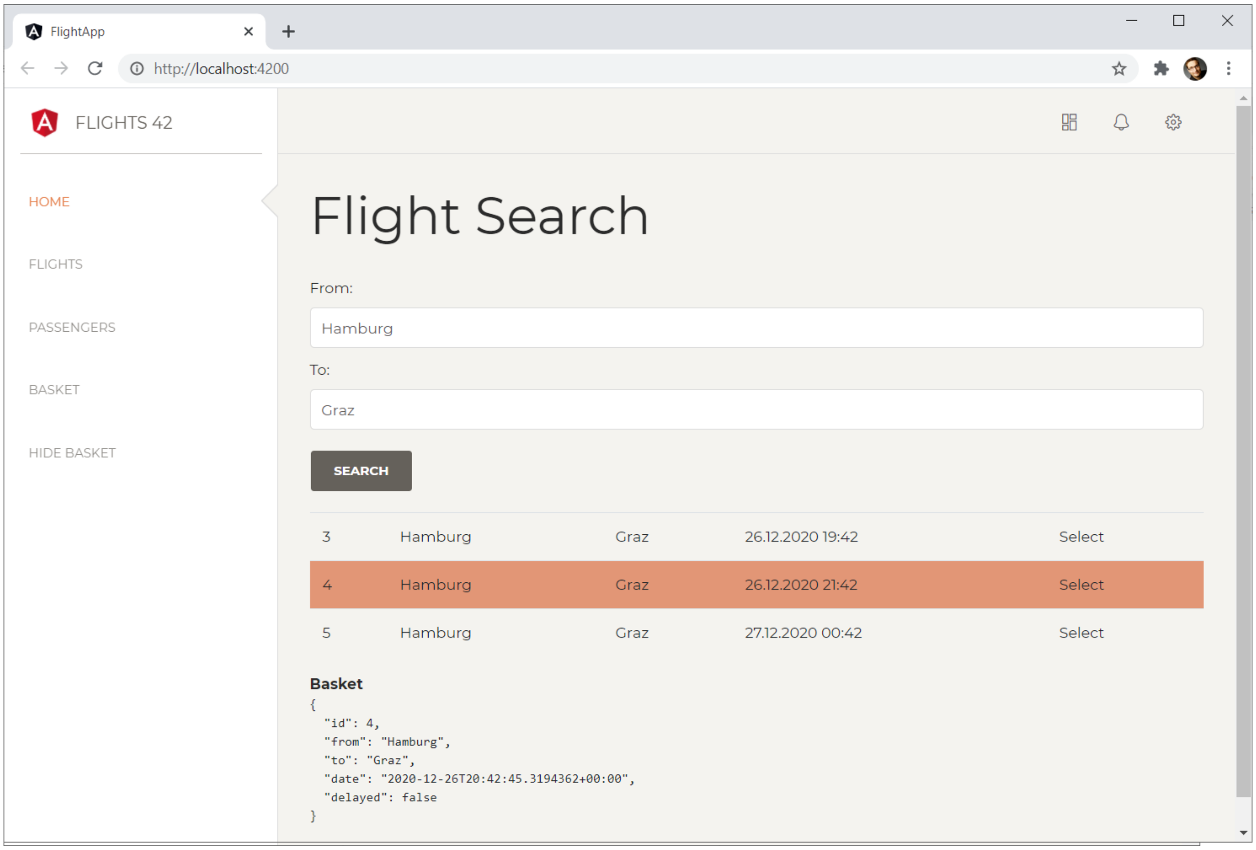
Anwendung starten

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

Zum Starten Ihrer Anwendung nutzen Sie die Angular CLI im Projekthauptverzeichnis:

```
ng serve -o
```

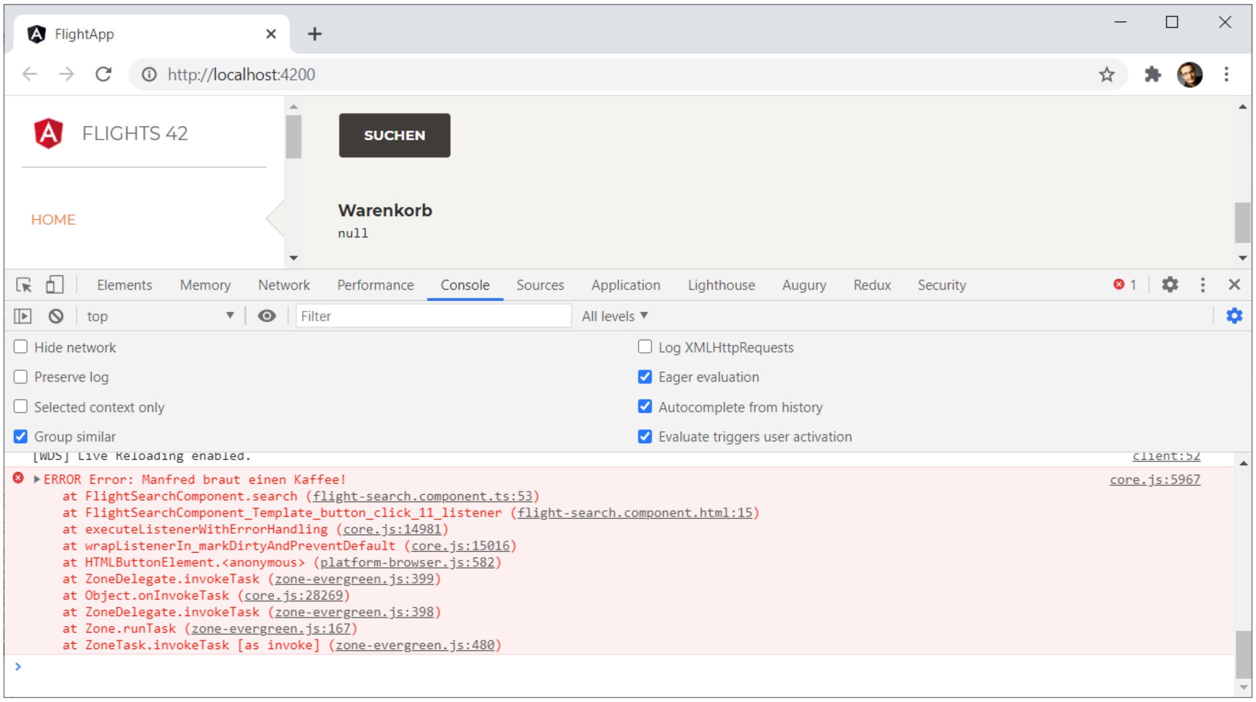
Nach dem Start des Entwicklungswebservers steht die Anwendung unter http://localhost:4200 bereit:



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:



Fehler in der Entwicklerkonsole

Der gezeigte Fehler wurde zur Veranschaulichung mit der Anweisung

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

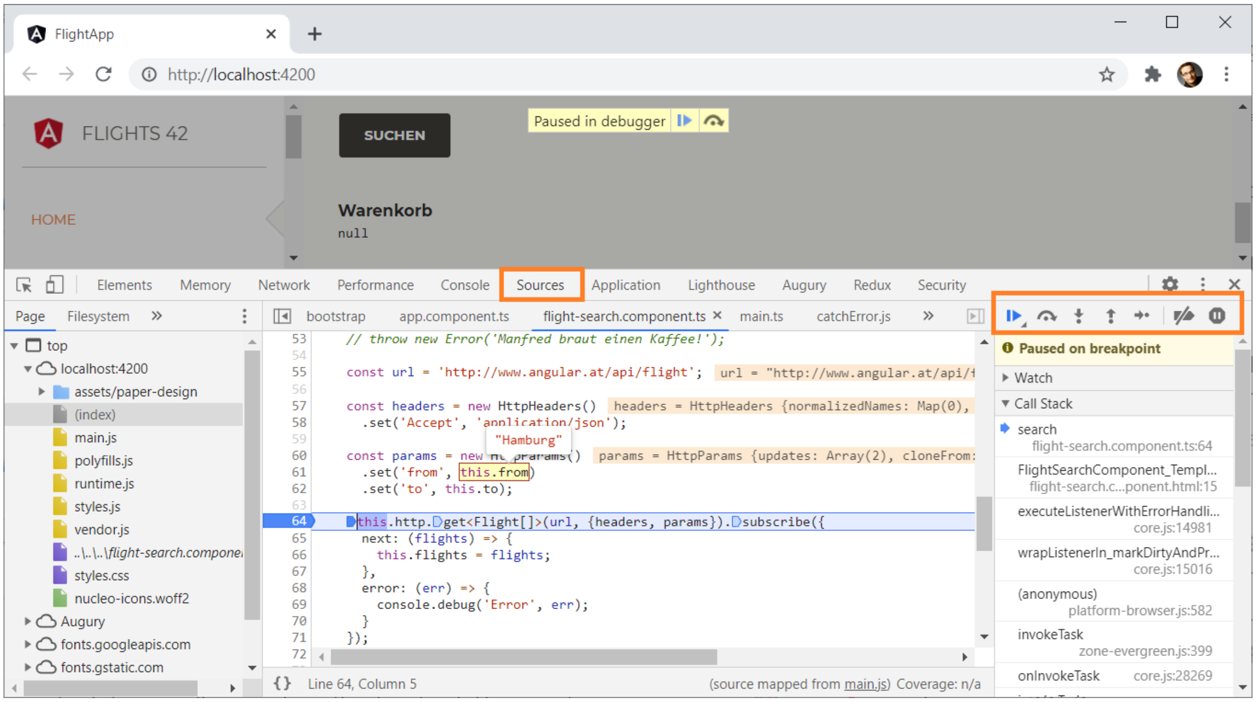
am Anfang der Methode `search` provoziert. In der Regel ist das jedoch nicht notwendig: Anwendungen weisen häufig auch ohne weiteres Zutun Bugs auf ;-).

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.

Bonus: 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*:



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

Bonus: 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:

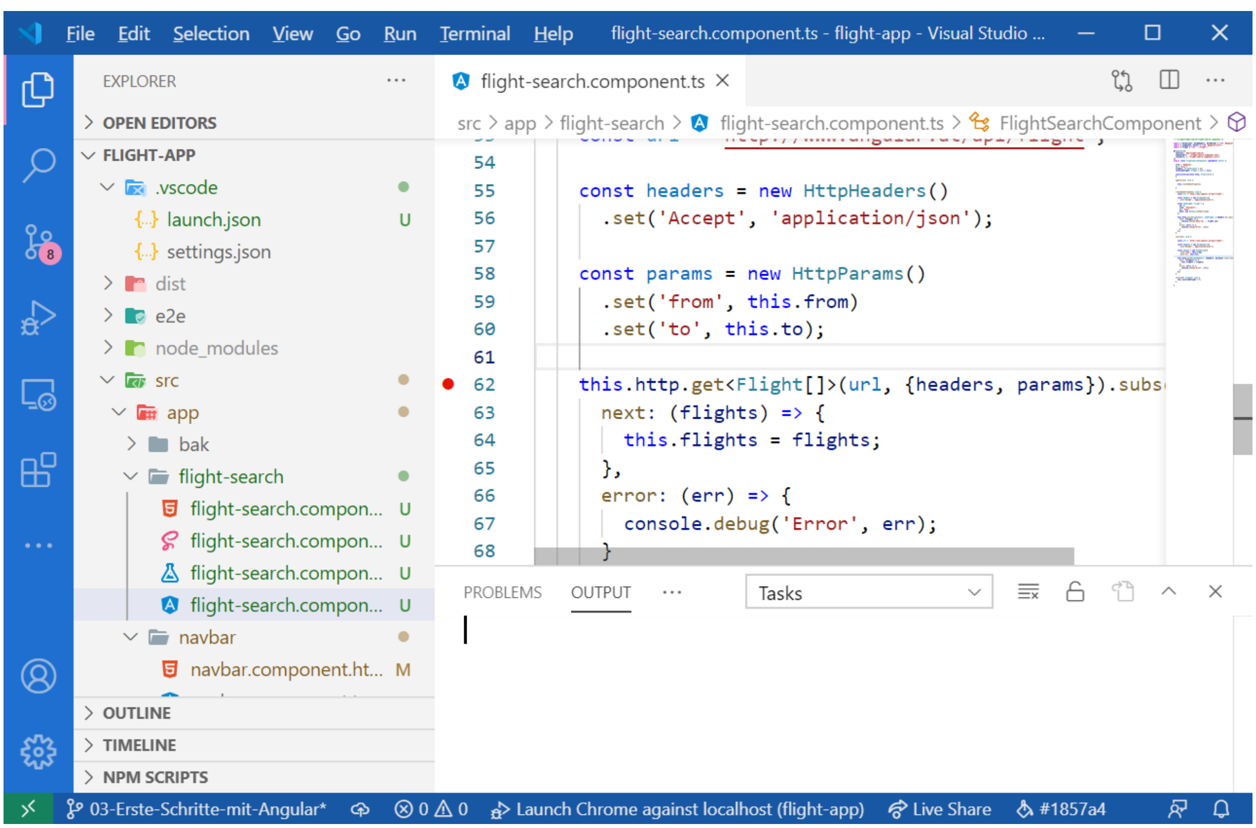
1. Öffnen Sie eine beliebige `.ts`-Datei.
2. Wählen Sie in Visual Studio Code den Befehl `Run/Start Debugging` oder drücken Sie `F5`.
3. Falls Visual Studio Code Sie nach einer Umgebung (Environment) für das Debugging fragt, wählen Sie *Chrome* aus.
4. Visual Studio Code generiert nun eine Datei `launch.json`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

```
{
    "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.

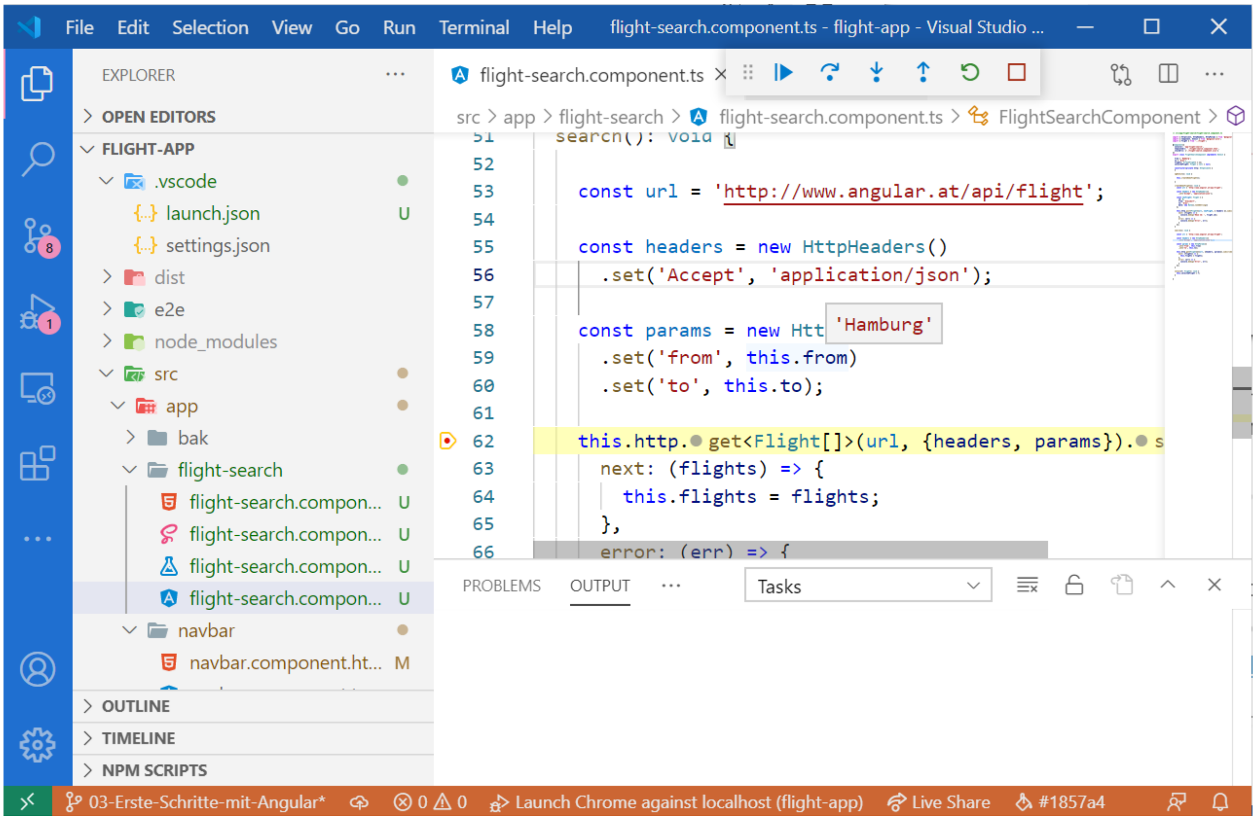
Um den Debugger nun via Visual Studio Code zu nutzen, sind die folgenden Schritte notwendig:

1. Starten Sie Ihre Anwendung wie gewohnt mit `ng serve`.
2. Erzeugen Sie direkt in Visual Studio Code durch einen Klick links neben eine Zeilennummer einen Break Point:



Break Point in Visual Studio Code (Zeile 62)

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.



Debuggen mit Visual Studio Code

Zusammenfassung

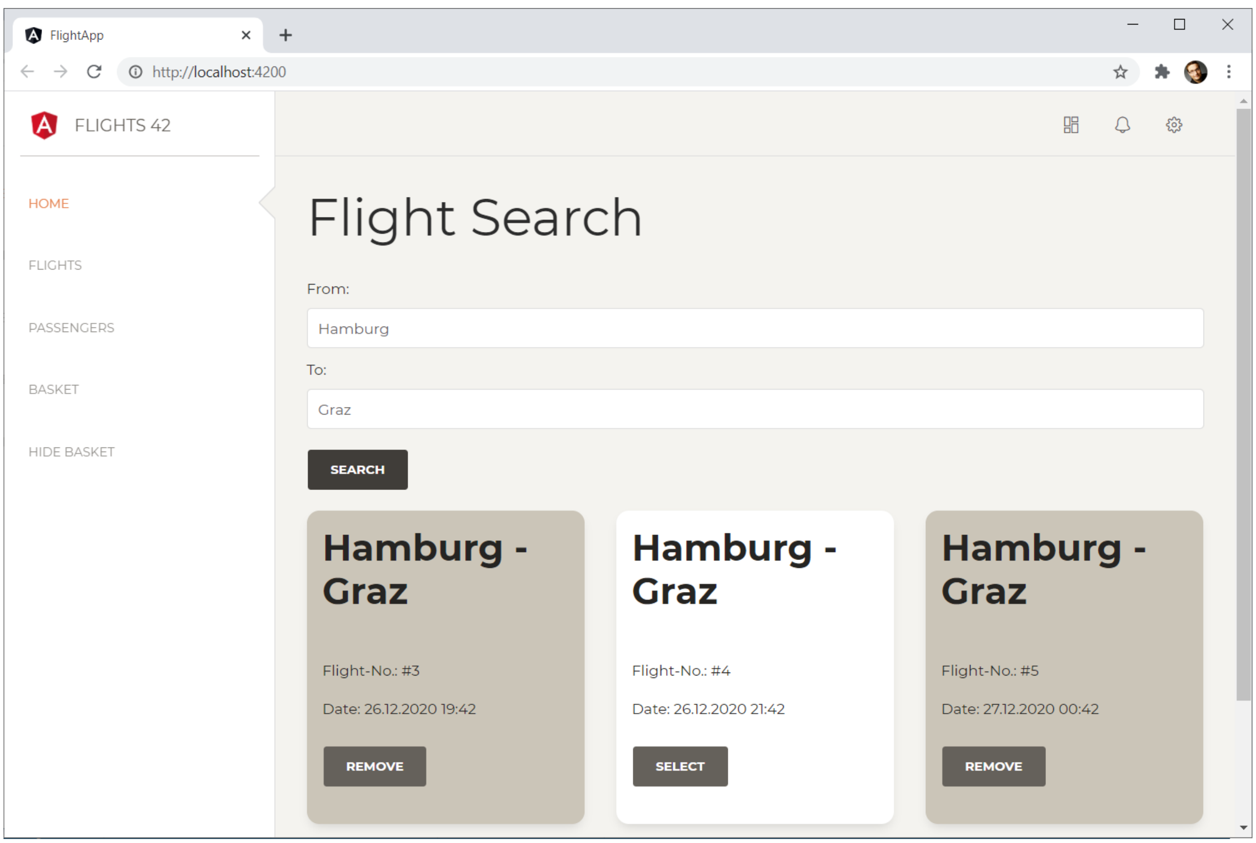
Angular-Anwendungen bestehen aus Komponenten. Hierbei handelt es sich um Klassen, die Informationen über Eigenschaften sowie das gewünschte Verhalten über Methoden anbieten. Dazugehörige Templates definieren, wie Angular die Komponenten darstellt. Mit Datenbindungsausdrücken stellen sie die Eigenschaften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

Wiederverwendbare Sub-Komponenten und Services

In diesem Kapitel wollen wir unsere Lösung mit Boardmittel von Angular ein wenig verfeinern. Dazu lagern wir wiederverwendbare UI-Fragmente in eine Sub-Komponente und eine wiederverwendbare Logik in einen Service aus.

Sub-Komponenten mit Event- und Property-Bindings

Jede Angular-Komponente kann weitere Komponenten in ihrem Template aufrufen. Zur Kommunikation kommen dazu die aus dem letzten Kapitel bekannten Property- und Event-Bindings zum Einsatz. Um dies zu veranschaulichen, kommt hier eine Sub-Komponente, die Flüge in Form von Karten präsentiert zum Einsatz:



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.

> 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
  }
  [...]
}
```

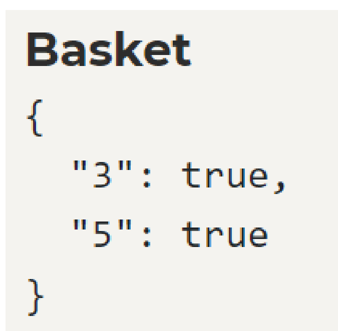
Um festzustellen, ob sich ein Flug im Warenkorb befindet, muss die Anwendung also nur prüfen, ob der Basket an der Stelle der *FlugId* *truthy* ist:

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

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

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

Das Ganze gestaltet sich dann, wie nachfolgend gezeigt:



Ausgabe des Warenkorbs

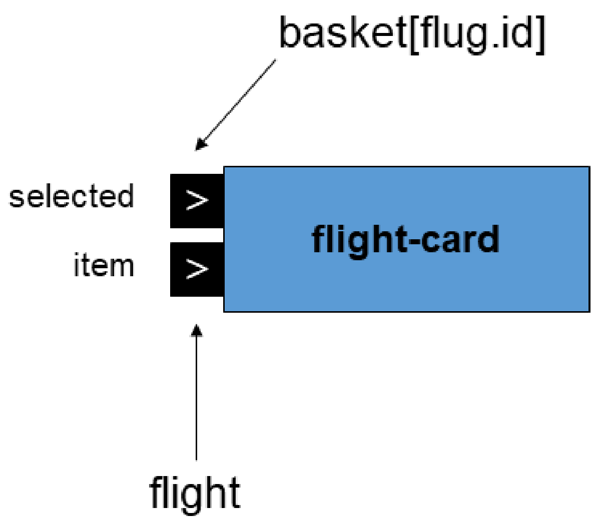
Eine Komponente mit Property-Bindings Property-Binding

Die hier besprochene Karte, deren Implementierung im nächsten Abschnitt folgt, soll über Property-Bindings zwei Informationen vom Parent übergeben bekommen: den anzuzeigenden 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:



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.

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 im letzten Kapitel 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:

```typescript
// 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:

```html
<!-- 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:

```scss
.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`. Normalerweise kümmert sich die CLI automatisch darum. Zur Sicherheit empfiehlt es sich jedoch, diesen Umstand zu prüfen:

```
// 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:

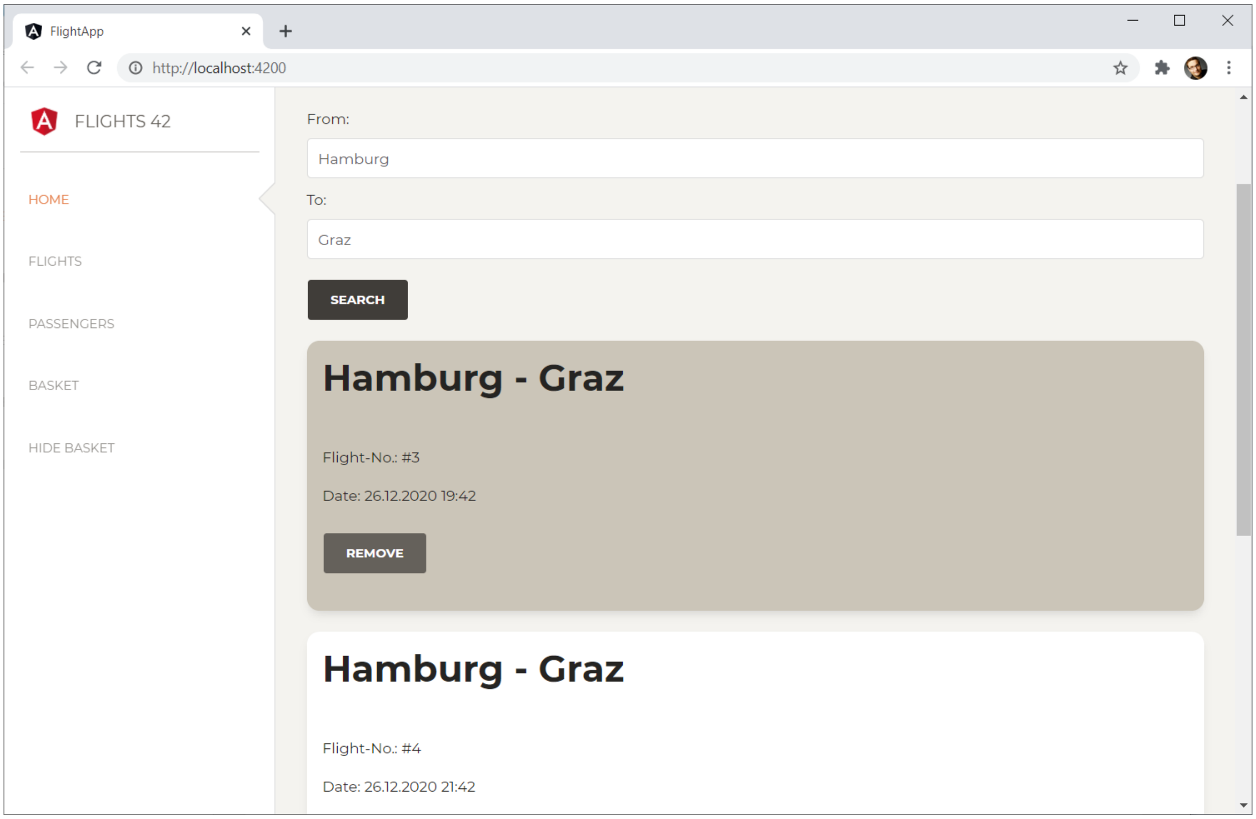
```
<div *ngFor="let f of flights">
    <app-flight-card [item]="f" [selected]="basket[f.id]">
    </app-flight-card>
</div>
```

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.

Bonus: Responsive Design mit dem Bootstrap Grid Layout

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 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.

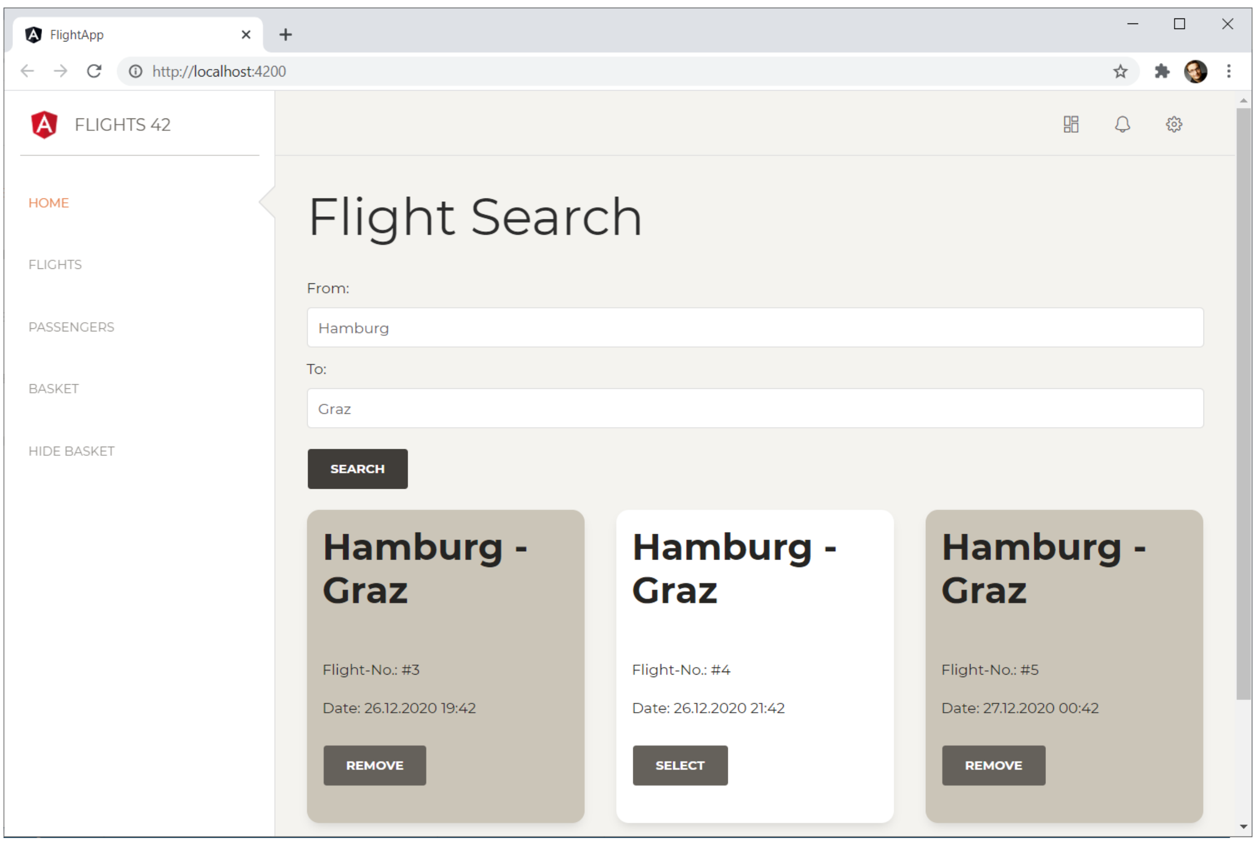
Beispielsweise könnte man nun angeben, dass eine Karte bei sehr kleinen Geräten (`xs`) alle zwölf Spalten erhält, bei kleinen (`sm`) sechs, bei mittleren (`md`) sowie bei großen (`lg`) vier und bei sehr großen (`lg` und `xl`) drei der insgesamt zwölf Spalten. Somit werden je nach Auflösung eine bis vier 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 Event-Binding

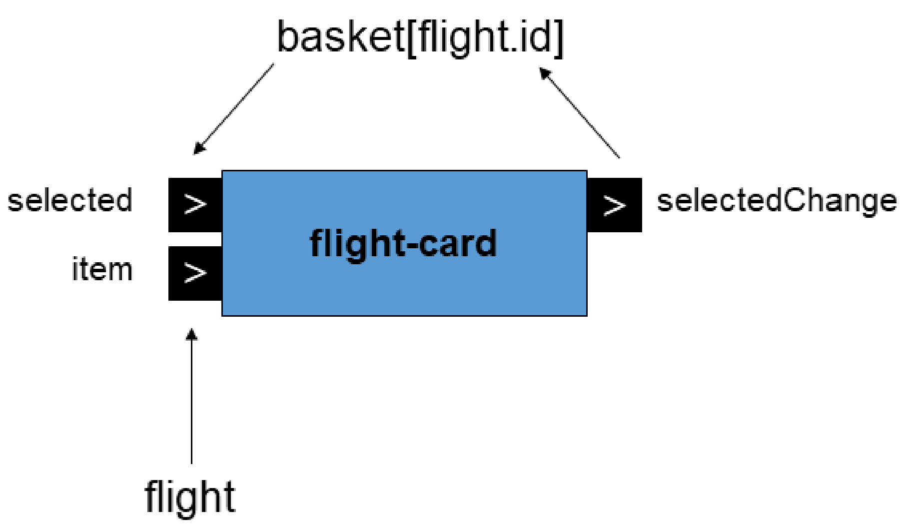
Dieser Abschnitt erweitert die hier gezeigte `FlightCardComponent` um ein Ereignis `selectedChange`. Dieses Ereignis soll den Parent informieren, wenn die Karte aus- bzw. abgewählt wird:

```
<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. Im hier betrachteten Fall fließt der Wert `selected` über einen Ausgang zurück in den Warenkorb:



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:

```
// 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.

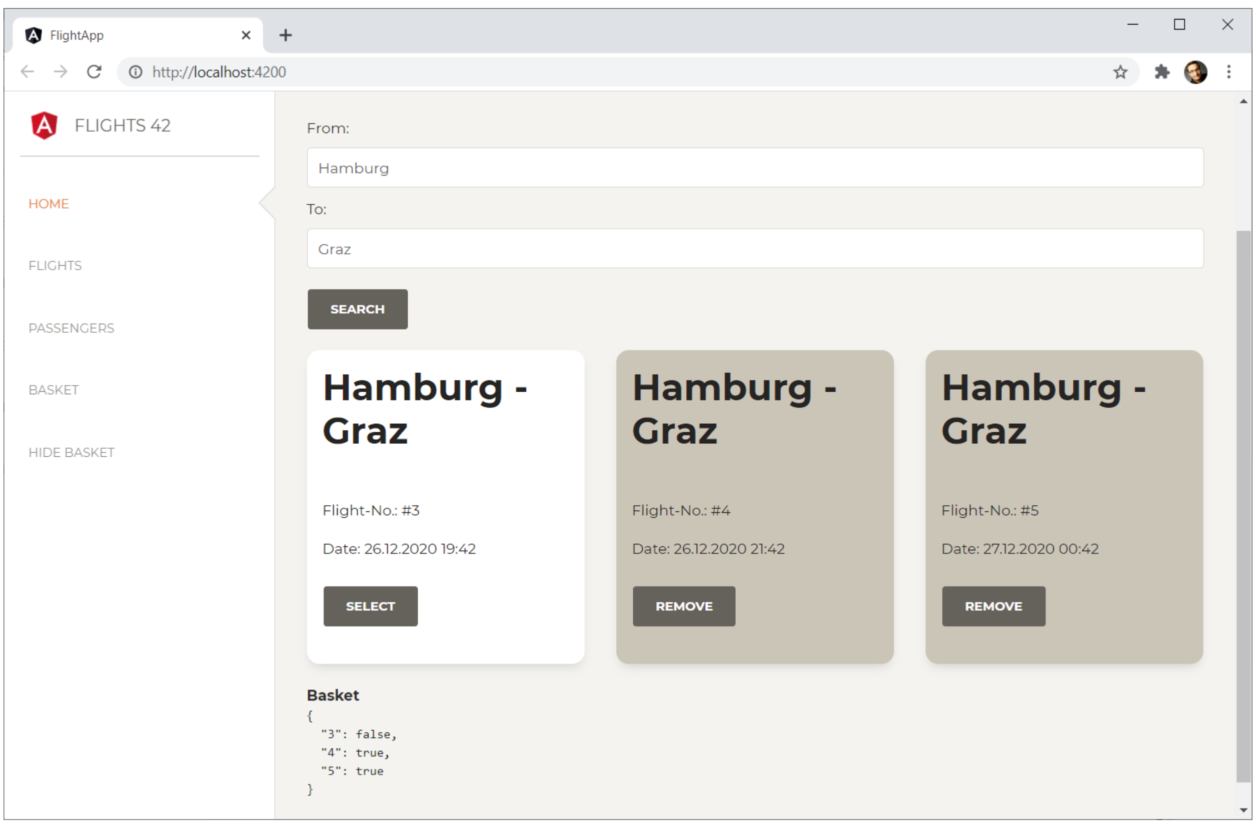
Damit der `EventEmitter` den neuen Wert von `selected` veröffentlichen kann, wird er mit Boolean typisiert.

Komponente aufrufen

Nach dieser Erweiterung können Sie mit dem Aufruf der `FlightCardComponent` einen Event-Handler für `selectedChange` festlegen:

```
<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:



Der Warenkorb wird nun aktualisiert.

Komponenten mit Two-Way-Bindings

Wir haben eine gute Nachricht: Unsere `Input`/`Output`-Kombination von `selected` und `selectedChange` erfüllt sämtliche Konventionen für die verkürzte Banana-in-a-Box-Schreibweise. 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:

```
<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>
```

Die Grammatik für Two-Way-Bindings ist tatsächlich nur eine Schreiberleichterung, die Angular in Fällen, wo die diese Konventionen erfüllt sind, ermöglicht.

Wiederverwendbare Logik in Services auslagern

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.

Dieser Abschnitt 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.

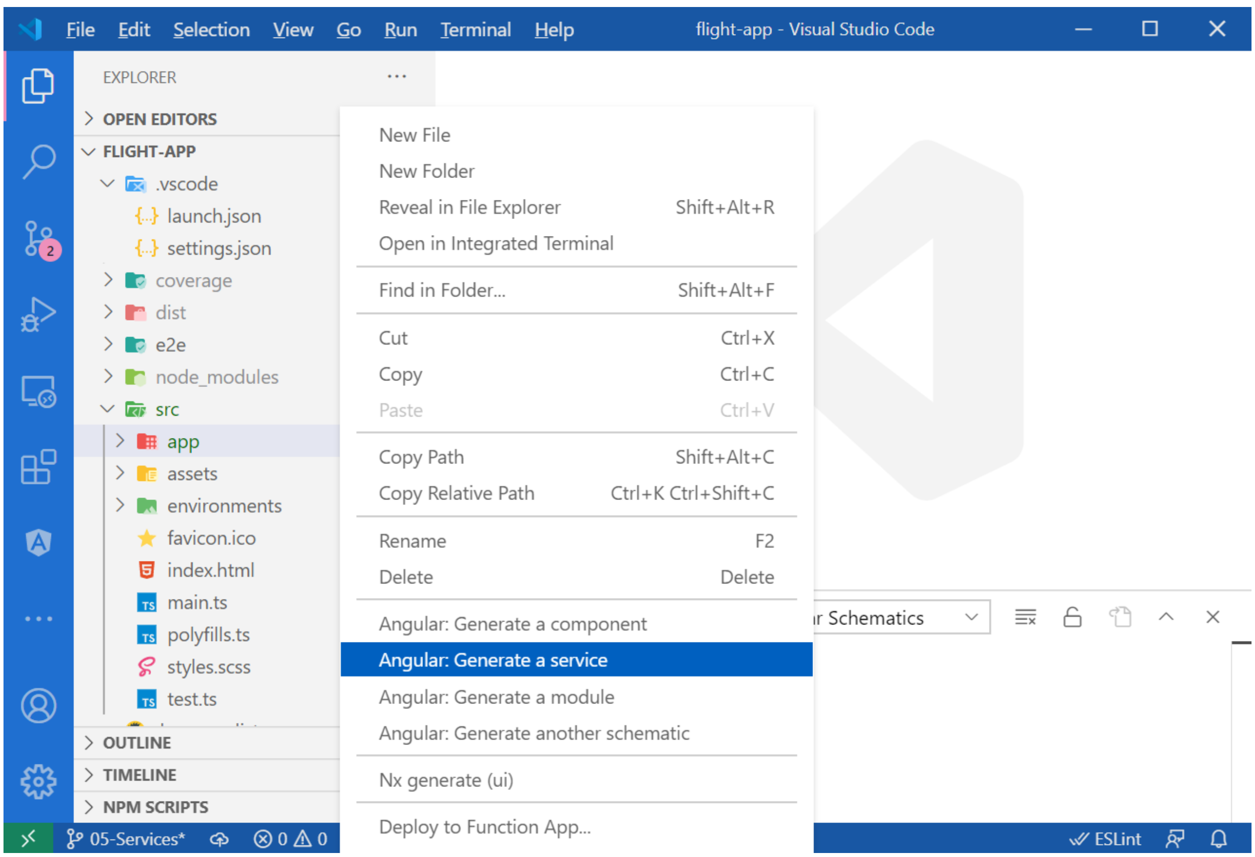
Genau das wird auch unsere erste Aufgabe in diesem Kapitel sein. Ähnlich wie Komponenten lassen sich Servicegenerieren generierenServices mit der Angular CLI generieren. Führen Sie dazu den folgenden Befehl im Hauptverzeichnis Ihres Projekts aus:

```
ng generate service flight
```

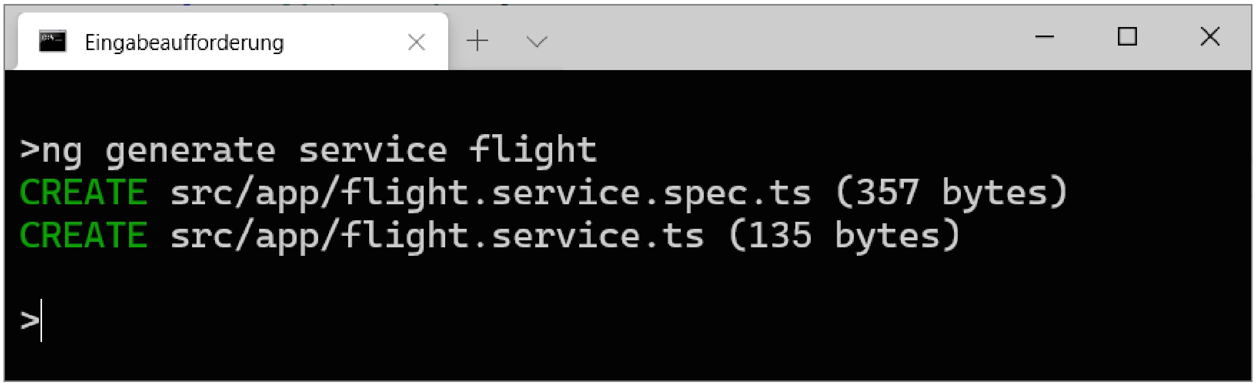
Die Anweisung zum Generieren eines Service lässt sich auch abkürzen:

```
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:



Dieser Befehl veranlasst die CLI, zwei Dateien zu generieren:



Service mit der CLI generieren

Die generierte Datei `flight.service.ts` enthält das Grundgerüst unseres neuen `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*Provider* oder *Serviceprovider*. Wird der Service wie hier über Eigenschaften von `Injectable` konfiguriert, ist auch von Tree-Shakable Provider die Rede. Der Name rührt 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 kleineren 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 `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.

  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:

```
// 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.

Den Service konsumieren

Nun können wir unseren `FlightService` in der `FlightSearchComponent` nutzen:

```
// 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;

    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.

Der zuvor injizierte `HttpClient` wird nicht mehr benötigt. Deswegen wurde seine Verwendung aus der `FlightSearchComponent` ersatzlos entfernt. Das betrifft auch die im letzten 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.

Zusammenfassung

Angular bietet einige Building-Blocks zur Schaffung wiederverwendbarer Anwendungsteile: (Sub-)Komponenten weisen wiederverwendbare UI-Fragmente auf und kommunizieren mit ihren Eltern-Komponenten über Property- und Event-Bindings. Diese sind mit den Dekoratoren `@Input` und `@Output` zu kennzeichnen. Services kapseln hingegen wiederverwendbare Logiken und lassen sich in andere Services und Komponenten injizieren.

Navigationsstrukturen schaffen: Der Angular Router

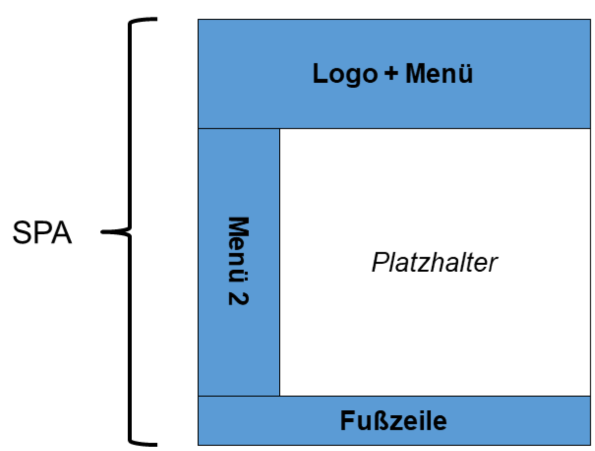
Eine *Single Page Application* (SPA) besteht, wie der Name schon ausdrückt, aus nur einer Seite. Um verschiedene Anwendungsfälle anbieten zu können, müssen wir verschiedene Seiten simulieren. Das erfolgt durch das Ein- und Ausblenden von Komponenten. Der Angular-Router hilft bei dieser Aufgabe.

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

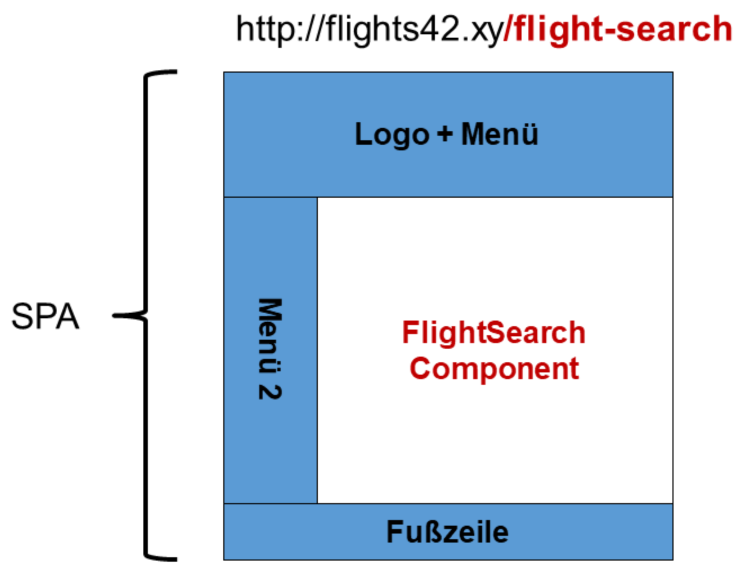
Wenn eine SPA mehrere Seiten simulieren soll, reicht es nicht, einfach nur Komponenten ein- und auszublenden. Damit der Back-Button so funktioniert, muss sich der durchgeführte Zustandswechsel in der URL widerspiegeln. Dasselbe gilt für Bookmarks oder Links, die auf eine bestimmte Ansicht der SPA verweisen. Glücklicherweise automatisiert der Router auch diese Aufgabe, die man ebenfalls als *Deep Linking* 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:



SPA mit Platzhalter für das Routing

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:

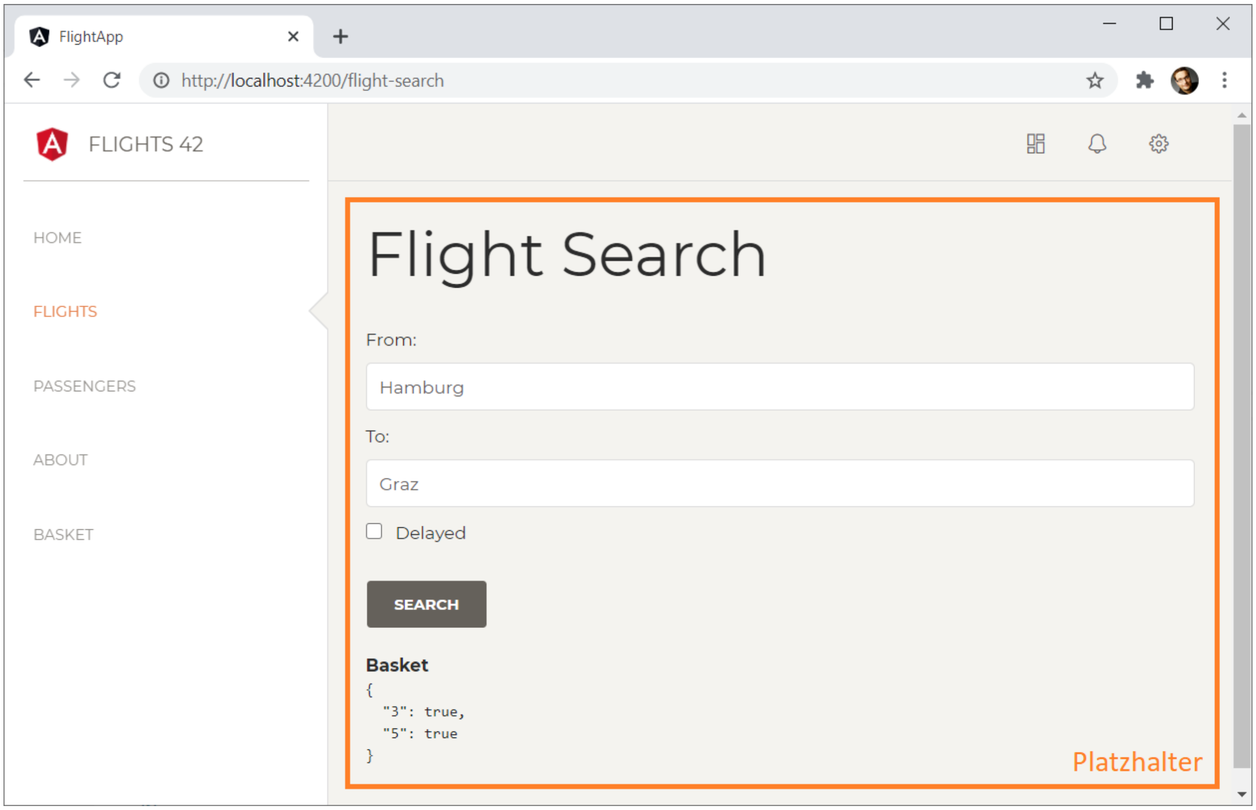


Aktivieren von Komponenten mit dem Angular-Router

Hier wurde an die URL der SPA der Pfad */flug-suchen* angehängt. Das veranlasst den Router, die damit assoziierte `FlightSearchComponent` zu aktivieren.

Komponenten für das Routing einrichten

Um die Funktionsweise des Routers zu veranschaulichen, werden wir endlich die Menübefehle auf der linken Seite an unsere Bedürfnisse anpassen und mit Leben erfüllen:



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. Vorerst handelt es sich dabei lediglich um eine Dummy-Komponente, die als weiteres Routing-Ziel fungiert.
- `FlightEditComponent`: Komponente zum Editieren von Flügen. Auch hierbei handelt es sich um eine Dummy-Komponente, die als weiteres Routing-Ziel zum Einsatz kommt. Anders als bei der `PassengerSearchComponent` nehmen wir hier allerdings einen Routing-Parameter entgegen.
- `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 passenger-search
ng generate component flight-edit
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` 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 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.

Bei einer RouterKonfiguration Konfiguration 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:

```
// src/app/app.routes.ts

import { Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { FlightSearchComponent } from './flight-search/flight-search.component';
import { PassengerSearchComponent }
    from './passenger-search/passenger-search.component';
import { FlightEditComponent } from './flight-edit/flight-edit.component';
import { AboutComponent } from './about/about.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: 'flight-search',
        component: FlightSearchComponent
    },
    {
        path: 'flight-edit/:id',
        component: FlightEditComponent
    },
    {
        path: 'passenger-search',
        component: PassengerSearchComponent
    },
    {
        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.

Eventuell haben Sie auch die Endung `:id` im Pfad der `FlightEditComponent` entdeckt. Hierbei handelt es sich um einen Platzhalter mit dem Namen id. Den übergebenen Wert können wir später in der `FlightEditComponent` auslesen.

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 `RouterModule` ins `AppModule` zu 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.

Platzhalter in AppComponent hinterlegen

Anstatt auf eine konkrete Komponente zu verweisen, nutzt die `AppComponent` nun einen Platzhalter. Dieser repräsentiert der Router durch ein `router-outlet`-Element:

```html
<!-- 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>
```

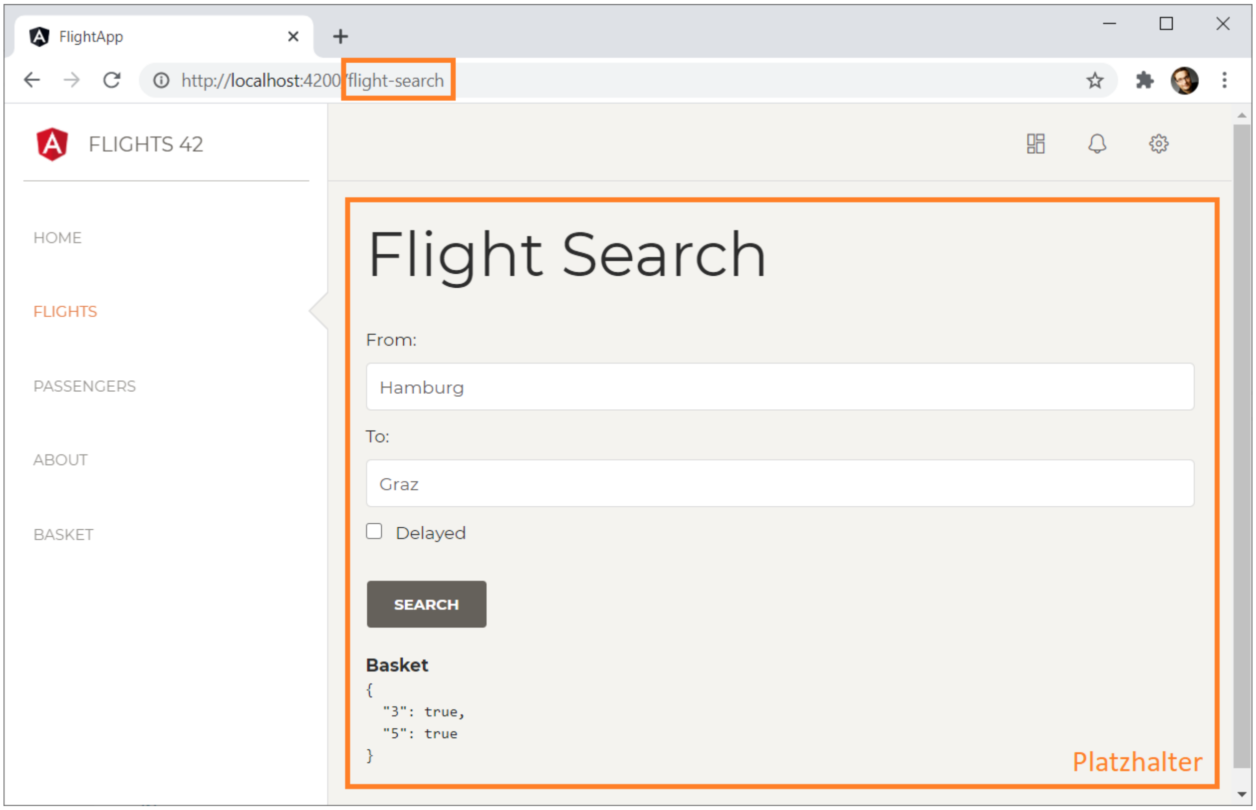
Hyperlinks zum Aktivieren von Routen nutzen

Nun benötigen wir nur noch Hyperlinks, die die einzelnen Routen im Platzhalter aktivieren. Dazu passen wir die generierte `SidebarComponent` an:

```html
<!-- 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 `routerLink` aufweist oder dies auf ein Child-Element zutrifft.

Wenn Sie nun Ihre Anwendung starten, sollten die Menüeinträge auf der linken Seite auf die einzelnen Routen verweisen:



Routing in der Demoanwendung

Die aktuelle Route wird nun auch durch die URL in der Adresszeile widergespiegelt.

Routen-Parameter auslesen

Um den für `flight-edit` konfigurierten Routing-Parameter auslesen zu können, fordert die `FlightEditComponent` den Service `ActivatedRoute` an:

```
// 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.

Bei `id` handelt es sich um jenen Parameter, den wir in der Routenkonfiguration vorgesehen haben. Den Parameter `showDetails` haben wir hingegen nicht konfiguriert. Aus diesem Grund geht Angular davon aus, dass er in Form eines Name/Wert-Paares an den Pfad angehängt wird:

```
/flight-edit/17;showDetails=true
```

Im letzteren Fall spricht der URL-Standard auch über Matrix-Parameter. Diese werden durch Strichpunkte getrennt und beziehen sich per Definition auf das letzte Url-Segment und bei Angular somit auf die damit assoziierte Komponente. Der besser bekannte Query-String, der nach einem Fragezeichen an die Url angehängt wird, bezieht sich hingegen per Definition immer auf die gesamte Url.

Das Template der Komponente präsentiert diese Eigenschaften:

```
<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. Allerdings könnte man die Informationen aus den letzten Kapiteln nutzen, um den Flug mit der erhaltenen Id zu laden und über ein Formular zum Editieren anzubieten.

Auf parametrisierte Routen verweisen

Um auf parametrisierte Routen zu verweisen, nimmt `routerLink` die einzelnen URL-Segmente, aber auch Matrixparameter als Array entgegen. Das nachfolgende Beispiel erweitert das Template der `FlightCardComponent` um `routerLink`, der zur zuvor eingeführten `FlightEditComponent` führt:

```
<!-- 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-SegmenteURL-Segment. 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`.

Programmatisch Routen

Statt mit Hyperlinks können Sie einen Routenwechsel auch programmatisch anstoßen. Lassen Sie sich dazu den `Router` injizieren:

```
[…]
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 Aufruf

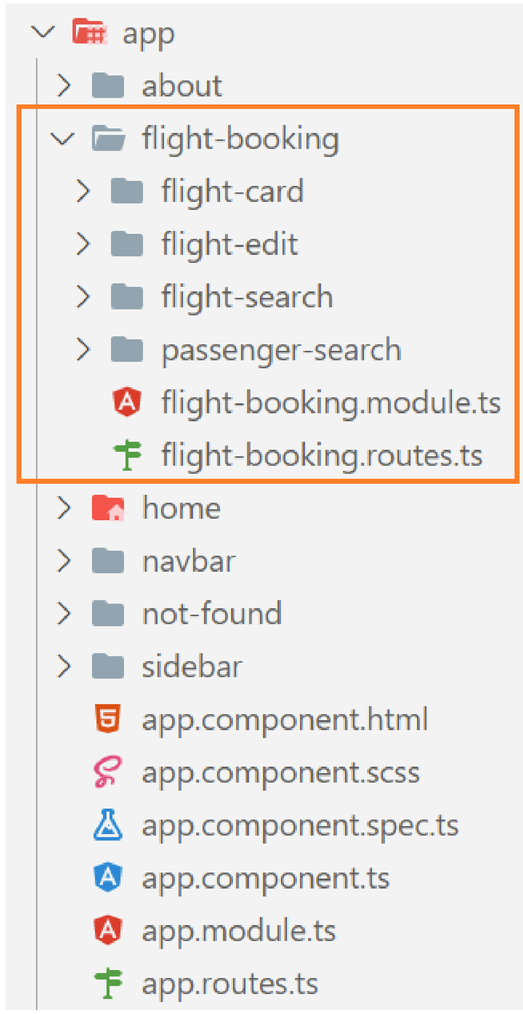
```
this.router.navigate(['/flight-edit', id]);
```

führt somit zur Aktivierung der Route */flight-edit/17*, wenn man davon ausgeht, dass die Variable `id` den Wert *17* hat.

Bonus: Routing und Module

Bis jetzt haben wir zur Vereinfachung nur ein einziges Modul, nämlich das `AppModule`, verwendet. Um die Anwendung besser zu strukturieren bietet es sich an, jedes einzelne Feature in ein eigenes Modul zu verschieben. Jedes dieser Module bekommt in der Regel einen eigenen Ordner aber auch eine eigene Routen-Konfiguration.

In unserem Fall bietet sich ein `FlightBookingModule` an:



Feature Module für Flight Booking

Sämtliche Komponenten unseres Flight Booking-Features wurden in den neuen `flight-booking` Ordner verschoben. Falls Sie das selbst ausprobieren sollten sie sicherstellen, dass nach dem Verschieben sämtliche Import-Anweisungen noch auf die korrekten Dateipfade verweisen.

> Da diese Vorgehensweise die Struktur unserer Anwendung ein wenig verändert, haben wir den Quellcode für dieses Bonus-Kapitel in einen eigenen Branch unseres Beispiel-Projektes[12] ausgelagert.

Im Ordner `flight-booking` finden wir auch die Routen-Konfiguration, die sich auf die Komponenten des Modules beschränkt:

[12] https://github.com/manfredsteyer/angular-intro/tree/modules

```
// src/app/flight-booking/flight-booking.routes.ts

import { Routes } from '@angular/router';
import { FlightEditComponent }
    from './flight-edit/flight-edit.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-search',
        component: FlightSearchComponent
    },
    {
        path: 'passenger-search',
        component: PassengerSearchComponent
    },
    {
        path: 'flight-edit/:id',
        component: FlightEditComponent
    },
];
```

Beim `FlightBookingModule` handelt es sich wie beim `AppModule` um eine Klasse mit Metadaten:

```
// src/app/flight-booking/flight-booking.module.ts

import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FlightSearchComponent }
    from './flight-search/flight-search.component';
import { FlightCardComponent }
    from './flight-card/flight-card.component';
import { PassengerSearchComponent }
    from './passenger-search/passenger-search.component';
import { FlightEditComponent }
    from './flight-edit/flight-edit.component';
import { RouterModule } from '@angular/router';
import { FormsModule } from '@angular/forms';

import { FLIGHT_BOOKING_ROUTES } from './flight-booking.routes';
```

```

@NgModule({
  imports: [
    CommonModule,

    // Important: Routes are referenced with **forChild**
    RouterModule.forChild(FLIGHT_BOOKING_ROUTES),

    // Dont't forget this if you want to work with Forms
    FormsModule,
  ],
  declarations: [
    FlightSearchComponent,
    FlightCardComponent,
    PassengerSearchComponent,
    FlightEditComponent,
  ],
})
export class FlightBookingModule { }
```

Wichtig ist hier, das die Routen für Feature-Module an `RouterModule.forChild` zu übergeben sind. Lediglich das Root-Modul, also unser `AppModule`, ruft `RouterModule.forRoot` auf. Das ist eine übliche Konvention in der Welt von Angular. Sie stellt sicher, dass eine Bibliothek wie der Router globale Services nur ein einziges mal via `forRoot` einrichtet. Bei jeder weiteren Verwendung richtet `forChild` nur mehr jene zusätzlichen Strukturen ein, die für die zusätzliche Nutzung im jeweiligen Feature-Module benötigt werden.

Außerdem importiert dieses Beispiel das `CommonModule`. Diese Modul kommt mit Direktiven wie `*ngIf` oder `*ngFor` sowie mit den üblichen Pipes wie `date` oder `json`. Da wir diese Konstrukte in der Regel brauchen, erhält in der Regel jedes weitere Modul diesen Import.

> Eventuell ist Ihnen aufgefallen, dass das `AppModule` ohne Importieren des CommonModules auskommt und hier trotzdem `*ngIf`, `*ngFor` sowie die erwähnten Direktiven die ganze Zeit zur Verfügung standen. Das liegt daran, dass das `BrowserModule`, das wir von Anfang an im `AppModule` importiert haben, sämtliche Inhalte des `CommonModules` bietet.

Damit Angular das neue `FlightBookingModule` berücksichtigt, müssen wir es noch ins `AppModule` importieren:

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

[...]

// Add this import:
import { FlightBookingModule }
    from './flight-booking/flight-booking.module';

@NgModule({
   imports: [
      RouterModule.forRoot(APP_ROUTES),
      HttpClientModule,
      BrowserModule,
      FormsModule,

      // Add import for Feature-Module:
      FlightBookingModule,
   ],
   declarations: [
      AppComponent,
      SidebarComponent,
      NavbarComponent,
      HomeComponent,
      AboutComponent,
      NotFoundComponent
   ],
   providers: [],
   bootstrap: [
      AppComponent
   ]
})
export class AppModule { }
```

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.

Nächste Schritte

Unser Angular-Buch bei O'Reilly

Falls Ihnen die Art und Weise, wie wir die Entwicklung mit Angular in diesem Buch erklären, werden Sie auch unser "großes" Angular-Buch bei O'Reilly mögen:

Angular Buch bei O'Reilly

Es liegt mittlerweile in der 3. Auflage vor. Alle Details finden sich hier[13].

[13] https://oreilly.de/produkt/angular-2/

Trainings und Consulting

Erfahren Sie mehr über Angular für große Unternehmens-Anwendungen in unserem Advanced Online Workshop[14]:

Advanced Angular Workshop

Sichern Sie sich Ihre Tickets jetzt für sich und Ihre Kollegen.

Darüber hinaus bieten wir folgende Themen als Teil unserer Schulungs- oder Beratungsworkshops an:

- Angular Workshop: Strukturierte Einführung
- Advanced Angular: Enterprise Solutions und Architektur
- Professional Angular Testing Workshop (Cypress, Just, etc.)
- Reaktive Architekturen mit Angular (RxJS and NGRX)
- Angular Review Workshop
- Angular Upgrade Workshop

Wenn Sie Fragen haben, können Sie gerne auf uns zukommen: office@softwararchitekt.at[15].

Bleiben Sie mit uns in Kontakt, z. B. via Twitter[16] oder Facebook[17].

[14]https://www.angulararchitects.io/en/angular-workshops/advanced-angular-enterprise-architecture-incl-ivy/
[15]mailto:office@softwararchitekt.at
[16]https://twitter.com/manfredsteyer
[17]https://www.facebook.com/manfred.steyer