Angular

Das Praxisbuch – von den Grundlagen bis zur professionellen Entwicklung mit Signals

5 Komponenten und Signals: die Grundbausteine

Wir wollen mit Angular starten! In diesem Kapitel beschäftigen wir uns ausführlich mit Komponenten. Außerdem lernen wir den Grundbaustein *Signal* kennen und betrachten die Bestandteile der Template-Syntax. Im darauffolgenden Praxiskapitel werden wir das Wissen schließlich in der Beispielanwendung einsetzen.

5.1 Komponenten

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

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

Technisch ist eine Komponente immer eine TypeScript-Klasse, die mit dem Decorator `@Component()` versehen ist. Darüber werden verschiedene Metadaten an die Klasse angehängt, unter anderem können wir in der Eigenschaft `template` das Template für die Komponente definieren. Das Listing 5.1 zeigt den Grundaufbau einer Komponente.

Listing 5.1 *Eine simple Komponente*

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

5.2 Das Template einer Komponente

Das Template ist der sichtbare Teil der Komponente, mit dem wir in der Oberfläche interagieren können. Für die Beschreibung wird in der Regel HTML verwendet[1], denn wir wollen unsere Anwendung ja im Browser ausführen. In den Templates wird eine Angular-spezifische Syntax eingesetzt, denn Komponenten können weit mehr, als nur statisches HTML darzustellen. Diese Syntax schauen wir uns im Verlauf dieses Kapitels noch genauer an.

Um eine Komponentenklasse mit ihrem Template zu verknüpfen, gibt es zwei Wege:

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

Welchen dieser beiden Wege wir wählen, hängt vom persönlichen Geschmack und von der Größe des Templates ab. Für kleine Templates kann ein Inline Template lohnenswert sein – so hat man alle Dinge sofort auf einen Blick verfügbar. Wird das Template zu lang, empfehlen wir hingegen, es in eine eigene Datei auszulagern. Die Angular CLI legt das Template standardmäßig auch in einer separaten Datei ab. Dafür verwenden wir die Eigenschaft `templateUrl` im Decorator `@Component()`. In Listing 5.2 sind beide Varianten zur Veranschaulichung aufgeführt. Die gezeigte Kombination funktioniert natürlich nicht, denn eine Komponente hat immer genau ein Template.

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

Listing 5.2
Template einer Komponente definieren

```
@Component({
  // Referenz auf ein HTML-Template
  templateUrl: './my-component.html',
  // ODER: HTML-String direkt im TypeScript
  template: `<h1>Hello Angular!</h1>`,
})
export class MyComponent { }
```

Template und Komponentenklasse sind eng miteinander verknüpft und können über klar definierte Wege miteinander kommunizieren. Der Informationsaustausch findet über sogenannte *Bindings* statt, die wir gleich noch genauer betrachten.

5.3 Komponenten in der Anwendung verwenden

Jede Angular-Anwendung besteht aus mindestens einer Komponente. Die erste Komponente trägt in der Regel den Namen `App` und wird auch *Hauptkomponente* oder *Root Component* genannt. Diese Komponente ist dauerhaft sichtbar und zeigt ihre Inhalte in der Seite an.

Bei der Arbeit mit Angular entwickeln wir verschiedene weitere Komponenten, die jeweils einen kleinen Teil der Oberfläche beschreiben. Damit diese Komponenten sichtbar werden, müssen sie gezielt in die Anwendung eingebunden werden. Zu diesem Zweck besitzt jede Komponente einen Selektor, der im Property `selector` angegeben wird: Er legt fest, in welchen DOM-Elementen eine Instanz dieser Komponente erzeugt wird.

Um also eine Komponente einzubinden, müssen wir in einem Template der Anwendung ein Element anlegen, das zum Selektor der gewünschten Komponente passt. Platzieren wir z. B. das Element `<my-component />` im Template der Komponente `App`, so erzeugt Angular dort eine Instanz der oben gezeigten `MyComponent`. Dabei können wir ein Self-Closing Tag verwenden (`<my-component />`) oder das Element explizit schließen (`<my-component></my-component>`).

Ein solches Element wird als *Host-Element* bezeichnet: Es beherbergt eine Instanz der Komponente, mitsamt ihrem Template und ihrer Logik.

Auf diese Weise entsteht ein Baum von Komponenten: Ausgehend von der Wurzel `App` werden Komponenten eingebunden, die weitere Komponenten einbinden usw. Im folgenden Beispiel besitzt die Komponente `Sidebar` den Selektor `app-sidebar`. Platzieren wir ein DOM-Element mit diesem Namen in der Komponente `App`, entsteht dort eine Navigationsleiste, so wie sie in der Komponente `Sidebar` definiert ist. Wir

sprechen dabei von Eltern- und Kindkomponenten: `App` wird die Elternkomponente für ihre eingebundenen Kinder `Sidebar` und `MainArea`.

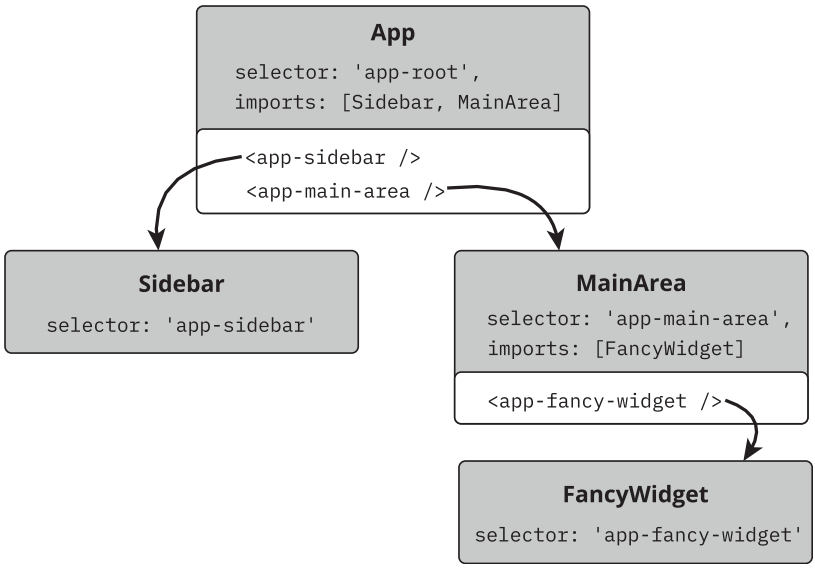
Damit diese Mechanik funktioniert, müssen wir in der Elternkomponente bekannt machen, welche Kindkomponenten im Template verwendet werden sollen. Der Decorator `@Component()` besitzt dafür das Feld `imports`: Hier müssen wir alle Komponenten (und auch Pipes und Direktiven) eintragen, die wir im Template dieser Komponente nutzen wollen. In unserem Beispiel bedeutet das: `App` muss `Sidebar` und `MainArea` importieren, um sie nutzen zu können.

Listing 5.3 *Komponenten importieren und einbinden*

```
import { Component } from '@angular/core';
import { Sidebar } from './sidebar/sidebar';
import { MainArea } from './main-area/main-area';

@Component({
  selector: 'app-root',
  imports: [Sidebar, MainArea],
  template: `
    <h1>My App</h1>
    <app-sidebar />
    <app-main-area />
  `
})
export class App {}
```

Abb. 5.1 *Komponentenbaum*



In den Templates der Kindkomponenten können wir weitere Komponenten einbinden. In Abbildung 5.1 wird `FancyWidget` so zur Kindkomponente von `MainArea`. Damit das funktioniert, sind in `MainArea` wieder die gleichen Schritte notwendig:

- Die Klasse `FancyWidget` muss unter `imports` eingetragen werden.
- Im Template muss das Element `<app-fancy-widget />` platziert werden.

In den meisten Fällen erzeugen wir die Host-Elemente für unsere Kindkomponenten selbst. Später im Kapitel zu Routing ab Seite 167 lernen wir einen weiteren Weg kennen, um Komponenten einzubinden: Der Router von Angular setzt Komponenten abhängig von der angefragten URL in die Anwendung ein.

Vielleicht ist dir in den Beispielen aufgefallen, dass die Selektoren immer mit dem Präfix `app` beginnen, z. B. `app-root` bei der Hauptkomponente. Damit vermeiden wir Konflikte mit nativen DOM-Elementen oder Komponenten aus anderen Projekten. Wir können das Präfix beim Anlegen des Projekts festlegen (`ng new --prefix=abc`) oder es später bei Bedarf in der Datei `angular.json` ändern.

5.4 Komponenten generieren mit der Angular CLI

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

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

```
$ ng generate component main-area
```

Listing 5.4
Komponente generieren mit der Angular CLI

Die Komponente wird in einem eigenen Unterordner mit den vier dazugehörigen Dateien generiert: TypeScript-Klasse, HTML-Template, Stylesheet und eine Testdatei mit der Endung `.spec.ts`. In diesem Beispiel wird also die Komponente `MainArea` erzeugt.

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

Listing 5.5
Komponente generieren mit Kurzbefehl

```
$ ng generate component main-area
$ ng g component main-area
$ ng generate c main-area
$ ng g c main-area
```

Es gibt übrigens keinen Befehl, um eine generierte Komponente wieder zu löschen. Wenn wir eine falsche Komponente angelegt haben, müssen wir die erzeugten Dateien manuell entfernen.

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

Listing 5.6
Dry Run für generate*-Befehle*

```
$ ng g component main-area --dry-run
```

Dieses Feature ist vor allem bei der Arbeit in großen Projekten wertvoll. Bevor wir etwas generieren, können wir uns so vergewissern, dass wir alle Parameter korrekt gesetzt haben. Wenn du dir unsicher bist, was ein Befehl genau tun wird, verwende bitte zuerst den Dry Run und prüfe die Ausgabe.

5.5 Reaktivität mit Signals

Eine große Stärke von Angular sind reaktive Datenbindungen: In den Templates unserer Komponenten wollen wir Daten und Werte anzeigen, die wir in der TypeScript-Klasse definiert haben. Ändern sich die Werte, soll das Template entsprechend aktualisiert werden.

Um diese Mechanik kümmert sich Angular automatisch im Hintergrund.[2] Damit das Framework weiß, wann sich ein Wert tatsächlich geändert hat, verwenden wir die sogenannten *Signals*.

Ein Signal ist eine *Reactive Primitive*, also ein Grundbaustein für Angular-Apps. Es ist ein Objekt, das einen Wert besitzt. Im Gegensatz zu einer einfachen Variable bzw. einem Property einer Komponente informiert das Signal alle Interessierten darüber, dass sich der Wert geändert hat. So kann das Framework gezielt alle Stellen aktualisieren, an denen der Wert benötigt wird, z. B. im Template einer Komponente.

Um ein Signal zu erstellen, verwenden wir die gleichnamige Funktion `signal()`. Hier müssen wir immer einen Startwert übergeben. Der Typ

[2] Der Prozess, um Templates bei Datenänderungen zu aktualisieren, heißt *Change Detection* bzw. *Synchronisation*.

wird automatisch ermittelt, komplexere Datentypen können wir in den spitzen Klammern explizit angeben. Das Ergebnis ist ein Objekt vom Typ `WritableSignal<T>`, wobei `T` den Typ des enthaltenen Werts angibt.

Listing 5.7
Ein Signal erstellen

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

interface MyData {
  name: string;
  // ...
}

@Component({ /* ... */ })
export class MyComponent {
  // WritableSignal<number>
  protected counter = signal(0);

  // WritableSignal<MyData>
  protected data = signal<MyData>({
    name: 'Angular', /* ... */
  });
}
```

Der Styleguide von Angular empfiehlt, den Access Modifier `protected` zu verwenden, wenn das Property im Template der Komponente verwendet wird. Das betrifft vor allem Propertys, die Signals beinhalten – es ist sehr wahrscheinlich, dass wir diese Daten später im Template anzeigen möchten. Wir haben diese Konvention im Grundlagenkapitel zu Syntax & Konzepten ab Seite 40 bereits erläutert.

Um den Wert eines Signals zu lesen, muss das Objekt wie eine Funktion aufgerufen werden. Es gibt dann synchron den aktuellen Wert zurück. Wir können ein Signal überall lesen, wo wir den Wert benötigen, z. B. in Methoden der Komponentenklasse oder im Template. Im folgenden Listing sehen wir bereits, wie wir mithilfe der Interpolation Daten im Template darstellen können – dazu gleich mehr.

Listing 5.8
Ein Signal im Template lesen

```
<p>Counter: {{ counter() }}</p>
<p>Name: {{ data().name }}</p>
```

Um den Wert zu aktualisieren, bietet ein `WritableSignal` die Methoden `set()` und `update()` an. Mit `set()` kann der Wert direkt überschrieben werden. `update()` führt eine Aktualisierung auf Basis des aktuellen Werts

durch. Die übergebene Arrow-Funktion erhält als Argument den derzeitigen Wert des Signals. Ihr Rückgabewert wird als neuer Wert in das Signal geschrieben.

Listing 5.9
Ein Signal updaten

```
this.counter.set(1); // 1
this.counter.update(c => c + 1); // 2
this.data.set({ name: 'Modern Angular', /* ... */ });
```

Signals sind notwendig, damit Angular gezielt auf Änderungen an unseren Daten reagieren kann. Für alle Daten, die wir im Template anzeigen möchten und die sich zur Laufzeit ändern können, müssen wir deshalb Signals verwenden. Für rein statische Werte, die sich niemals ändern, können wir auch einfache Propertys einsetzen.

Merke: Propertys, die wir im Template nutzen wollen und die sich ändern könnten, sollten immer als Signal angelegt werden.

5.6 Template-Syntax

Grundsätzlich können wir in den Templates ohne Einschränkungen alle Features und Tags von HTML verwenden. Damit wir aber nicht nur statisches HTML definieren können, erweitert Angular die gewohnte Syntax mit einer Reihe von Ausdrücken. Auf diese Weise können wir dynamische Features direkt in den Templates nutzen: Ausgabe von Daten, Kontrollfluss, Reaktion auf Ereignisse und das Zusammenspiel von mehreren Komponenten mit Bindings. Wir stellen in diesem Abschnitt die Template-Syntax kurz vor, bevor wir in den folgenden Kapiteln noch ausführlicher auf alle wichtigen Konzepte eingehen werden.

{{ Interpolation }}

Ein zentraler Bestandteil der Template-Syntax ist die Interpolation. Damit können wir Daten im Template einer Komponente anzeigen. Wir setzen dafür zwei geschweifte Klammern und notieren dazwischen einen sogenannten *Template-Ausdruck*. Dieser Ausdruck bezieht sich immer direkt auf die zugehörige Komponentenklasse. Im einfachsten Fall ist ein solcher Ausdruck eine Referenz auf ein Signal in einem Property aus der Klasse – aber auch der Aufruf von Methoden ist grundsätzlich möglich. Wichtig: Ein Property oder eine Methode muss immer `public` oder `protected`

sein, damit sie im Template referenziert werden kann. Private Propertys können nicht im Template verwendet werden.

Template-Ausdrücke können auch komplexer sein und z. B. Arithmetik enthalten. Vor der Ausgabe wird jeder Ausdruck ausgewertet, und der Rückgabewert wird schließlich im Template angezeigt.

Listing 5.10 *Komponente mit Propertys und Methoden*

```
@Component({ /* ... */ })
export class MyComponent {
  protected readonly vatRate = 0.19;
  protected title = signal('Hello World');
  protected currentIndex = signal(0);

  getMyText() {
    return 'Lorem ipsum dolor sit amet.';
  }
}
```

Listing 5.11 *Interpolation verwenden*

```
<p>Einfaches Property: {{ vatRate }}</p>
<p>Signal: {{ title() }}</p>
<p>Methode: {{ getMyText() }}</p>
<p>Arithmetik: {{ currentIndex() + 1 }}</p>
```

Wir können auf diese Weise übrigens nur Inhalte anzeigen, die sich als String darstellen lassen. Binden wir mit der Interpolation etwas ein, das kein String ist, wird es in einen String umgewandelt. Bei Zahlen funktioniert das problemlos, bei einem Objekt wird aber z. B. immer der Text `[object Object]` ausgegeben – das ist nur wenig hilfreich. In der Regel zeigen wir also nie ein komplettes Objekt an, sondern einzelne Eigenschaften daraus. Um tatsächlich den Inhalt des Objekts als JSON darzustellen, bietet Angular die `JsonPipe` an, die wir im Kapitel zu Pipes ab Seite 289 kennenlernen werden.

Interaktion mit DOM-Elementen

Das Template wird mit HTML-Elementen aufgebaut, die im Browser zu DOM-Elementen gerendert werden. Diese Elemente bieten uns Schnittstellen zur Interaktion an: Wir können Daten an Elemente übergeben (*Propertys*) und Ereignisse aus Elementen empfangen und verarbeiten (*Events*).

Für beide Varianten der Interaktion stellt Angular eine eigene deklarative Syntax bereit, mit der wir sogenannte *Bindings* definieren können. Die so gebundenen Daten und Methoden werden stets aktualisiert, wenn

sich der Wert ändert bzw. ein Event ausgelöst wird. Diese Kommunikation ist schematisch in Abbildung 5.2 dargestellt.

Mit einem *Property Binding* (eckige Klammern) übergeben wir Daten von der Komponente an ein DOM-Element. Im folgenden Listing 5.12 setzen wir das Property `href` des Anker-Elements auf den Wert des Signals `myUrl`.

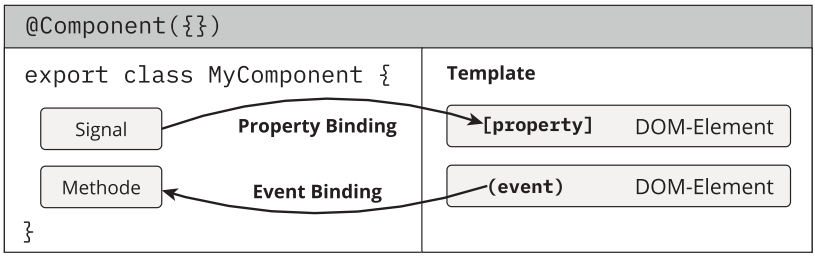
Ein *Event Binding* (runde Klammern) funktioniert genau umgekehrt: Es abonniert ein Event auf einem DOM-Element und transportiert die empfangenen Daten in die Komponentenklasse. Im Beispiel abonnieren wir das Event `click` auf dem Button und führen die Methode `myClickHandler()` in der Komponente aus.

Listing 5.12
Bindings im Template

```
<!-- Property Binding: Daten an DOM-Element übergeben -->
<a [href]="myUrl()">My Link</a>

<!-- Event Binding: Event empfangen und verarbeiten -->
<button type="button" (click)="myClickHandler()">
  Click me
</button>
```

Abb. 5.2
Bindings zwischen Komponente und Template



In den folgenden Kapiteln werden wir uns noch sehr ausführlich mit Bindings beschäftigen, denn sie sind die Grundlage für die Interaktion mit DOM-Elementen:

- Property Bindings und Component Inputs (ab Seite 97)
- Event Bindings und Component Outputs (ab Seite 117)

Eine Besonderheit ist das Two-Way Binding als syntaktische Kombination von Property Binding und Event Binding: Wir können damit eine Datenbindung in beide Richtungen gleichzeitig herstellen. Diese Variante wird

14 BookManager: Bücher lokal suchen

Mit dem erlernten Wissen zu den Konzepten rund um Signals wollen wir jetzt ein Computed Signal in der Praxis nutzen. Wir wollen dafür in der Buchliste eine Suchfunktion umsetzen. In der Liste werden dann nur die Bücher angezeigt, die zum eingegebenen Suchbegriff passen.

14.1 Praktische Umsetzung

Über der Buchliste in der `BooksOverviewPage` wollen wir ein Suchfeld platzieren. Wenn wir einen Suchbegriff in dieses Feld eingeben, soll die Buchliste lokal gefiltert werden, sodass nur die passenden Bücher angezeigt werden. Der Suchbegriff wird ebenfalls in einem Signal erfasst, das wir mit dem Suchfeld im Template synchronisieren müssen.

Die Filterung wollen wir mit einem Computed Signal `filteredBooks` implementieren: Es reagiert auf die Änderungen am Suchbegriff und gibt die gefilterte Liste aus. Im Template zeigen wir schließlich diese Auswahl anstelle der vollständigen Buchliste an.

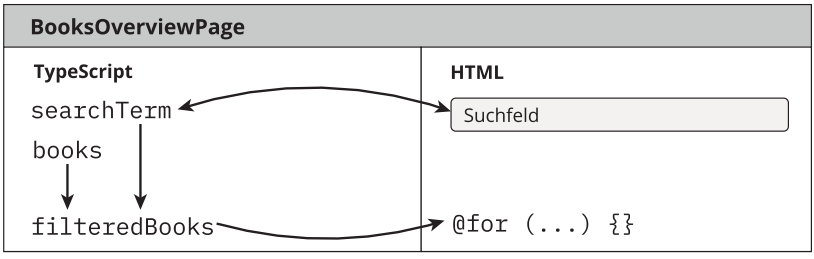


Abb. 14.1
Lokale Suche: Kommunikation in der BooksOverviewPage

Suchfeld anlegen

Zur Umsetzung der Suche benötigen wir im ersten Schritt ein weiteres Property in unserer Komponente `BooksOverviewPage`. In dem Signal `searchTerm` wollen wir gleich den aktuell eingegebenen Suchbegriff festhalten.

Listing 14.1 *Signal* searchTerm *anlegen* (books-overview-page.ts)

```
// ...
@Component({ /* ... */ })
export class BooksOverviewPage {
  #bookStore = inject(BookStore);

  protected searchTerm = signal('');
  protected books = signal<Book[]>([]);
  protected likedBooks = signal<Book[]>([]);
  // ...
}
```

Im nächsten Schritt benötigen wir ein Eingabefeld für die Suche. Dazu platzieren wir im Template ein `<input>`-Element mit dem Typ `search`, das wir mit dem Signal `searchTerm` verbinden wollen. Die Daten sollen in beide Richtungen fließen: Ändern wir die Eingabe im Formularfeld, soll das Signal aktualisiert werden. Ändert sich der Wert im Signal, soll dieser Wert im Formularfeld angezeigt werden. Genau genommen ist diese zweite Richtung für unseren Anwendungsfall gar nicht notwendig. Später werden aber noch weitere Anforderungen hinzukommen, sodass wir die Datenbindung gleich vollständig implementieren.

Wir verwenden ein Event Binding, um auf das native DOM-Event `input` zu reagieren. Das Event wird ausgelöst, sobald wir den Text im Eingabefeld ändern. Im Payload des Events befindet sich mit dem Property `target` eine Referenz auf das DOM-Element. Daraus können wir direkt den Wert lesen und so das Signal setzen.

Um den aktuellen Wert des Signals in das Feld zu schreiben, verwenden wir ein Property Binding auf das native Property `value`.

Listing 14.2 *Das Suchfeld im Template anlegen* (books-overview-page.html)

```
<!-- ... -->
<section>
  <h1>Books</h1>
  <div>
    <input type="search"
      [value]="searchTerm()"
      (input)="searchTerm.set($event.target.value)"
      placeholder="Search"
      aria-label="Search" />
```

```
    @for (b of books(); track b.isbn) {
      <app-book-card [book]="b"
        (like)="addLikedBook($event)" />
    }
  </div>
</section>
```

Buchliste filtern

Wir haben jetzt ein Eingabefeld und ein Signal, das immer den aktuellen Suchbegriff enthält. Außerdem haben wir die vollständige Liste der Bücher in dem Signal `books`. Nun müssen wir die Buchliste filtern, um die Suchergebnisse zu erhalten.

Dafür verwenden wir ein Computed Signal mit dem Namen `filteredBooks`. In der Funktion zur Berechnung prüfen wir zunächst, ob überhaupt ein Suchbegriff eingegeben wurde. Ist das nicht der Fall, geben wir die vollständige Liste der Bücher zurück. Wurde ein Begriff eingegeben, filtern wir die Liste der Bücher: Wir verwenden die Methode `filter()` auf dem Array, das wir aus dem Signal `books` erhalten. Enthält der Buchtitel den Suchbegriff, haben wir ein Ergebnis gefunden.

Damit die Suche nicht abhängig von Groß- und Kleinschreibung ist, wandeln wir die Strings vor dem Vergleich in Kleinbuchstaben um: Wir verwenden die Methode `toLowerCase()` für den eingegebenen Suchbegriff und den Titel des Buchs.

Das Computed Signal `filteredBooks` gibt nun also immer ein Array von Büchern zurück: entweder die gesamte Buchliste oder ein Array mit Ergebnissen zum eingegebenen Suchbegriff.

Listing 14.3
Computed Signal mit gefilterter Buchliste (books-overview-page.ts)

```
import { Component, computed, inject, signal } from
       ↪ '@angular/core';
// ...
@Component({ /* ... */ })
export class BooksOverviewPage {
  #bookStore = inject(BookStore);

  protected searchTerm = signal('');
  protected books = signal<Book[]>([]);
  protected likedBooks = signal<Book[]>([]);
```

```
  protected filteredBooks = computed(() => {
    if (!this.searchTerm()) {
      return this.books();
    }

    const term = this.searchTerm().toLowerCase();
    return this.books().filter((b) => b.title.toLowerCase()
        ↪ .includes(term));
  });
  // ...
}
```

Suchergebnisse anzeigen

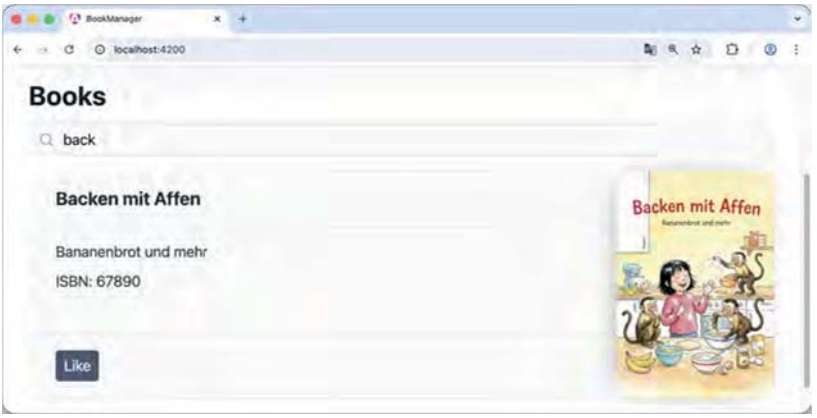
Zum Schluss müssen wir noch eine Anpassung im Template vornehmen. Anstatt die gesamte Buchliste anzuzeigen, wollen wir nur die Suchergebnisse in der Oberfläche darstellen. Deshalb ändern wir in der Schleife `@for` die zu iterierende Liste von `books` auf `filteredBooks`.

Listing 14.4 *Ergebnisse anzeigen* (books-overview-page.html)

```
<!-- ... -->
@for (b of filteredBooks(); track b.isbn) {
  <app-book-card [book]="b" (like)="addLikedBook($event)" />
}
```

Jetzt können wir in der Anwendung nach Büchern in der Liste suchen! Geben wir einen Suchbegriff ein, werden nur die Bücher ausgegeben, die zum Begriff passen.

Abb. 14.2 *Buchliste mit Suche*



14.2 Computed Signals testen

Die neue Suchfunktion basiert auf dem Signal `searchTerm` und dem Computed Signal `filteredBooks`. Mit ergänzenden Tests wollen wir sicherstellen, dass die Filterlogik korrekt arbeitet.

Wir wollen vier neue Testfälle entwickeln:

- Kein Suchbegriff eingegeben: Alle Bücher werden angezeigt.
- Suchbegriff »Affe«: Es erscheint nur das Buch mit passendem Titel.
- Groß-/Kleinschreibung wird beim Suchen ignoriert.
- Ein nicht existierender Suchbegriff liefert eine leere Liste zurück.

Signals arbeiten immer synchron, also muss unser Testsetup nicht weiter angepasst werden. Erneut umgehen wir die Sichtbarkeitseinschränkung (`protected`) über den direkten Zugriff mittels `component['property']`. Wir testen hier nur die Filterlogik selbst und betrachten nicht noch einmal das Rendering der `BookCard`.

Zur besseren Strukturierung der Tests verwenden wir das AAA-Pattern:

Arrange: Wir erzeugen die Instanz unserer Komponente und initialisieren sie vollständig über das `TestBed`. Dadurch steht das Signal `filteredBooks` sofort für unsere Tests zur Verfügung.

Act: Wir setzen das Signal `searchTerm` auf unterschiedliche Werte, um verschiedene Szenarien der Filterung zu simulieren. Da Signals synchron arbeiten, können wir nach dem Setzen eines Werts umgehend wieder das erwartete Ergebnis auslesen.

Assert: Anschließend prüfen wir jeweils, ob das Computed Signal `filteredBooks` den erwarteten Zustand annimmt. Wir prüfen, ob die Liste der gefilterten Bücher hinsichtlich Anzahl und Inhalt korrekt ist.

```
it('should display all books for empty search term', () => {
  component['searchTerm'].set('');

  const books = component['filteredBooks']();
  expect(books).toHaveLength(2);
});
```

Listing 14.5 *Unit-Test für das Signal* filteredBooks (books-overview-page.spec.ts)

```
it('should filter books based on the search term', () => {
  component['searchTerm'].set('Affe');

  const books = component['filteredBooks']();
  expect(books).toHaveLength(1);
  expect(books[0].title).toBe('Backen mit Affen');
});

it('should filter books ignoring case sensitivity', () => {
  component['searchTerm'].set('AFFEN');

  const books = component['filteredBooks']();
  expect(books).toHaveLength(1);
  expect(books[0].title).toBe('Backen mit Affen');
});

it('should return an empty array if no book matches', () => {
  component['searchTerm'].set('unbekannter Titel');

  const books = component['filteredBooks']();
  expect(books).toHaveLength(0);
});
```

Bei Bedarf können weitere Assertions ergänzt werden, zum Beispiel, um die ISBN oder den Untertitel zu prüfen.

Fazit

Signals arbeiten synchron und geben den aktualisierten Wert sofort aus. Das Testen von Signals ist daher sehr unkompliziert. Neben den grundlegenden Testfällen könnten weitere Edge Cases geprüft werden:

- Suchbegriffe mit Sonderzeichen oder Emojis,
- Suchbegriffe mit Umlauten,
- sehr lange Suchbegriffe,
- Filter nach mehreren Kriterien gleichzeitig oder
- Kombination von `computed()` mit weiteren Signals oder `effect()`.

Probiere diese Szenarien gern selbst aus!

Quelltext, Online-Demo und Differenzansicht:
https://bm1.angular-buch.com/#bm05

25 Formularverarbeitung mit Signal Forms

Formulare sind ein bewährter Ansatz für Dateneingaben in Webanwendungen. Das Spektrum reicht vom einfachen Suchfeld über Anmeldeformulare bis hin zu mehrstufigen Eingabestrecken. Dabei geht es um mehr als nur Textfelder: Es müssen Werte synchronisiert, Zustände verwaltet und Fehlermeldungen angezeigt werden.

Angular bietet mehrere durchdachte Ansätze zur Formularverarbeitung. In diesem Kapitel beschäftigen wir uns mit dem neuesten: Signal Forms.

25.1 Angulars Ansätze für Formulare

Die Grundlagen zu Formularen im Web sind zunächst nicht besonders kompliziert: Wir erstellen im HTML-Template eine Reihe von Eingabefeldern wie Textfelder, Dropdowns oder Checkboxen. Anschließend verarbeiten wir die Eingaben mit JavaScript. Dieser Ansatz kommt jedoch schnell an seine Grenzen, wenn es um komplexe Anforderungen und große Formulare geht:

- Die Daten zwischen Komponente und Template sollen in beide Richtungen ausgetauscht werden.
- Es soll visuelles Feedback passend zum Zustand ausgegeben werden.
- Es sollen Fehlermeldungen angezeigt werden.
- Das Formular soll auf Änderungen an den Eingabedaten reagieren.

Um diese Aspekte mit reinem JavaScript und HTML umzusetzen, ist viel manueller Code nötig: Jedes Element muss einzeln selektiert, verarbeitet, überprüft und ggf. zurückgesetzt werden.

Angular bietet deshalb komfortable Schnittstellen, mit denen die Eingabedaten zentral ausgewertet und verarbeitet werden können. So sind wir

in der Lage, Wert- und Zustandsänderungen zu überwachen und auf Änderungen im Formular zu reagieren. Zum Beispiel können wir das Absenden blockieren, solange die Eingaben ungültig sind. Fehlermeldungen unterstützen die Nutzerführung, sodass die Eingabe korrigiert werden kann. All diese Aspekte lassen sich mit den Formularansätzen von Angular bequem umsetzen.

Seit dem ersten Release von Angular existieren zwei verschiedene Lösungen für Formularverarbeitung: *Template-Driven Forms* und *Reactive Forms*. Ende 2025 wurde mit Angular 21 ein neuer Ansatz vorgestellt: *Signal Forms* bieten eine enge Integration mit Signals und vereinfachen die Arbeit mit reaktiven Daten. Wir werden uns in diesem Buch deshalb auf den neuen Ansatz konzentrieren.

Signal Forms: experimenteller Status

Zum Zeitpunkt der Veröffentlichung dieses Buchs gelten Signal Forms noch als *experimental*. Der gedruckte Stand kann deshalb von den stabilen Schnittstellen abweichen. Alle Änderungen, die wir in diesem Buch nicht berücksichtigen konnten, haben wir in einem Online-Kapitel zusammengefasst:
https://angular-buch.com/material/signal-forms

Reactive Forms und Template-Driven Forms

Die älteren Ansätze bleiben weiterhin verfügbar und werden in vielen Bestandsprojekten genutzt. Wenn du einen Einstieg in eine der beiden etablierten Formulartechnologien benötigst, empfehlen wir einen Blick in unsere Online-Kapitel:

- **Reactive Forms:**
  https://angular-buch.com/material/reactive-forms
- **Template-Driven Forms:**
  https://angular-buch.com/material/template-forms

Solange die bisherigen Ansätze offiziell unterstützt werden und nicht als »deprecated« gelten, ist es nicht notwendig, bestehende Anwendungen sofort umzustellen. Sollte eine Migration jedoch notwendig sein, stehen dem keine größeren Hürden im Weg: Signal Forms wurden bewusst kompatibel gestaltet. Für neue Formulare empfehlen wir, direkt mit Signal Forms zu starten – langfristig wird dieser Ansatz der Standard sein.

Prinzipien von Signal Forms

Bevor wir in die praktische Arbeit einsteigen, lohnt sich ein Blick auf die Designprinzipien von Signal Forms. Sie helfen dabei, die Architektur zu verstehen und Entscheidungen im Code nachzuvollziehen.

- **Volle Kontrolle des Datenmodells:** Die Formulardaten werden als Signal verwaltet, das wir selbst erstellen und jederzeit aktualisieren können.
- **Deklarative Logik:** Die Validierungslogik wird in einem Schema beschrieben.
- **Strukturelle Abbildung:** Die Feldstruktur wird direkt aus der Datenstruktur abgeleitet.
- **Interoperabilität:** Signal Forms sind kompatibel mit bestehenden Formularlösungen.

25.2 Datenmodell und Feldstruktur

Mit einem Formular erfassen wir Daten über verschiedene Eingabeelemente. Diese Daten werden in einem Signal gespeichert, das wir selbst definieren: das *Datenmodell*. Damit Angular die zugehörigen Zustände, Metadaten und Regeln verwalten kann, leiten wir aus dem Signal ein *Formularmodell* ab.

Datenmodell erzeugen

Im ersten Schritt müssen wir ein Datenmodell für das Formular erstellen. Dazu erzeugen wir ein Signal, das die Struktur des Formulars mit den Anfangswerten enthält: das *Datenmodell*. Es ist die zentrale Datenquelle für das Formular und wird später automatisch mit den Formulareingaben synchronisiert.

Bei der Definition des Datenmodells gilt es einiges zu beachten. Die Propertys sollten genau zu den vorhandenen Eingabeelementen passen – ungenutzte Propertys verkomplizieren die Verarbeitung. Auch die richtigen Typen sind wichtig, z. B. `string` für Texteingaben, `number` für Zahlenwerte oder `boolean` für Checkboxen. Verschachtelungen mit Objekten und Arrays sind möglich und können zur Gruppierung eingesetzt werden.

Vorhandene Entitäten der Anwendung erfüllen diese Anforderungen aber nicht immer. Dann ist es sinnvoller, einen eigenen Typ für das Formular zu definieren. Bei einem Registrierungsformular benötigen wir z. B. zwei Passwortfelder (`pw1` und `pw2`). Das zugrunde liegende Datenobjekt für Personen besitzt hingegen viele weitere Eigenschaften, aber nur ein einziges Property für das Passwort. In diesem Szenario wäre es aufwendig, das Datenobjekt passend zu machen – mit einem eigenen Typ hingegen sieht das Datenmodell einfach aus und ist genau auf das Formular angepasst:

Listing 25.1 *Datenmodell für ein Registrierungsformular*

```
interface RegisterFormData {
  username: string;
  password: {
    pw1: string;
    pw2: string;
  };
  emails: string[];
}

@Component({ /* ... */ })
export class RegisterForm {
  protected readonly registerModel =
    signal<RegisterFormData>({
      username: '',
      password: { pw1: '', pw2: '' },
      emails: ['']
    });
}
```

Bei der Typisierung des Signals stehen uns zwei Möglichkeiten zur Verfügung: TypeScript kann den Typ automatisch aus dem Startwert ableiten (Type Inference), oder wir definieren einen separaten Typ – etwa ein Interface oder einen Type-Alias. Im gezeigten Beispiel haben wir uns für ein eigenes Interface `RegisterFormData` entschieden. Technisch arbeitet der Compiler mit beiden Varianten gleich gut. Es ist also eine Frage des Codestils und der fachlichen Anforderungen, wie der Typ des Datenmodells definiert wird.

Formularmodell erzeugen

Das Datenmodell enthält nur die reinen Werte. Für Zustände und Regeln benötigen wir zusätzlich ein Formularmodell: Es vermittelt zwischen dem HTML-Formular und dem Datenmodell.

Um ein Formularmodell zu erstellen, verwenden wir die Funktion `form()`: Sie erzeugt aus dem Signal ein Objekt vom Typ `FieldTree`. Dieses Objekt besitzt die gleiche grundlegende Struktur wie unser Datenmodell, verwaltet aber zu jedem Knoten die zugehörigen Zustände. Die Struktur unseres Datenmodells bleibt davon unberührt, und wir können zu jedem Zeitpunkt die aktuellen Daten im zugrunde liegenden Signal abfragen oder ändern.

Listing 25.2 *Formularmodell anlegen*

```
import { form } from '@angular/forms/signals';

// ...
@Component({ /* ... */ })
export class RegisterForm {
  protected readonly registerModel =
    signal<RegisterFormData>({ /* ... */ });

  protected readonly registerForm = form(this.registerModel);
}
```

Mit dem `FieldTree` können wir durch die Struktur des Formulars navigieren: Wir können einzelne Felder erreichen und uns durch verschachtelte Objekte und Arrays bewegen. Jeder Knoten im Objekt ist wieder ein `FieldTree`, der einen Teilbaum oder ein Feld des Formulars beschreibt.

Ein `FieldTree` ist gleichzeitig eine Funktion, die wir aufrufen können, um einen `FieldState` zu erhalten. Dieses Objekt beschreibt die Zustände eines Formularknotens – dazu gleich mehr.

Listing 25.3 *Durch den Formularbaum navigieren*

```
const userField = this.registerForm.username;
// FieldTree<string, string>

const pw1Field = this.registerForm.password.pw1;
// FieldTree<string, string>

const userState = this.registerForm.username();
// FieldState<string, string>

const emailsState = this.registerForm.emails();
// FieldState<string[], string>
```

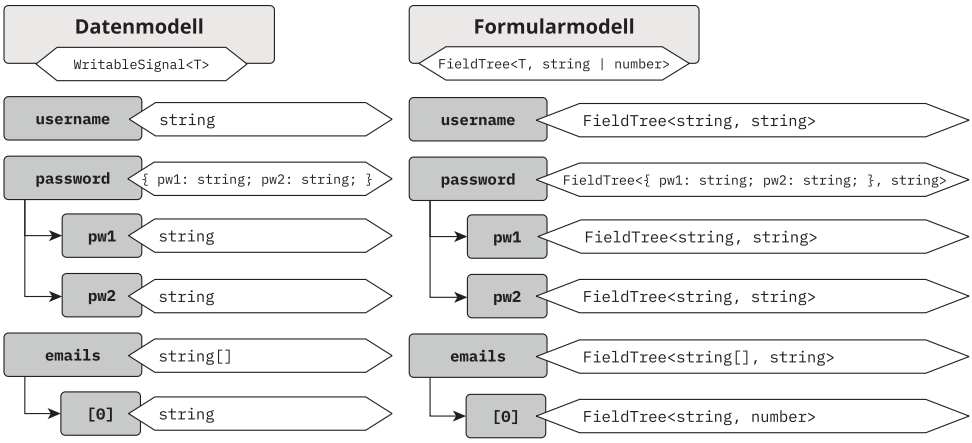


Abb. 25.1
Abbildung des Datenmodells auf das Formularmodell

Merke: Mit dem `FieldTree` **navigieren** wir durch das Formularmodell, um Teile oder einzelne Felder zu erreichen. Rufen wir das Objekt als Funktion auf, erhalten wir einen `FieldState`, der den **Zustand** des Knotens beschreibt.

- `FieldTree` → Navigation
- `FieldState` → Zustand

25.3 Formularmodell mit dem Template verknüpfen

Wir haben gesehen, wie ein Datenmodell angelegt und daraus ein Formularmodell abgeleitet wird. Jetzt verbinden wir die Struktur mit den HTML-Eingabefeldern.

Dafür verwenden wir die Direktive `FormField`, die wir in den Imports der Komponente eintragen:

Listing 25.4
Direktive FormField *importieren*

```
import { FormField } from '@angular/forms/signals';

@Component({
  // ...
  imports: [FormField],
})
export class RegisterForm {
  protected readonly registerModel = // ...
  protected readonly registerForm = // ...
}
```

Die Direktive hat die Aufgabe, eine Verbindung zwischen den Eingabefeldern und dem Formularmodell herzustellen. Das funktioniert mit allen üblichen HTML-Eingabeelementen wie `<input>`, `<textarea>` und `<select>`. An die Direktive übergeben wir immer einen `FieldTree` aus dem Formularmodell. So weiß Angular, welcher Teil der Formularstruktur zu welchem HTML-Eingabeelement gehört.

Listing 25.5
Formular mit dem Template verknüpfen

```
<form>
  <label>Username
    <input type="text" [formField]="registerForm.username" />
  </label>
  <label>Password
    <input type="password"
      [formField]="registerForm.password.pw1" />
  </label>
  <label>Password Confirmation
    <input type="password"
      [formField]="registerForm.password.pw2" />
  </label>
  <fieldset>
    <legend>E-Mails</legend>
    @for (email of registerForm.emails; track email) {
      <label>E-Mail {{ $index + 1 }}
        <input type="email" [formField]="email" />
      </label>
    }
  </fieldset>
</form>
```

Bei den Passwortfeldern `pw1` und `pw2` ist gut zu erkennen, dass auch Felder aus verschachtelten Objekten verwendet werden können. Beim Array `emails` iterieren wir mit `@for` direkt über den `FieldTree` aus `registerForm.emails` und erhalten so Zugriff auf die einzelnen E-Mail-Felder.

25.4 Werte und Zustände verarbeiten

Mit dem Objekt `FieldState` erhalten wir Zugriff auf den Zustand eines Feldknotens. Zur Erinnerung: Wir erhalten den `FieldState`, indem wir einen `FieldTree` aufrufen.

Die Eigenschaft `value` ist ein Signal, das wir lesen und schreiben können. So ist es also möglich, den Wert eines einzelnen Felds oder Formular-

knotens zu verarbeiten. Das Datenmodell im Signal und die Werte im Formularmodell werden stets synchronisiert.

Listing 25.6 *Werte und Zustände verarbeiten*

```
// FieldState
const formState = this.registerForm();
const usernameState = this.registerForm.username();

// Zugriff auf reaktive Zustände
const formValid = formState.valid();
const usernameTouched = usernameState.touched();

// Werte lesen (Formularmodell oder Datenmodell)
const username1 = this.registerForm.username().value();
const username2 = this.registerModel().username;

// Werte setzen
this.registerForm.username().value.set('MyUser');
this.registerForm.emails().value.update(
  emails => [...emails, 'team@angular-buch.com']
);
```

Diese Synchronisierung funktioniert in beide Richtungen: Wenn wir die Werte in unserem Datenmodell aktualisieren, werden diese automatisch im `FieldState` aktualisiert. Um einen Wert programmatisch zu schreiben oder zu lesen, ist es also egal, ob wir dafür das Datenmodell oder das Formularmodell verwenden.

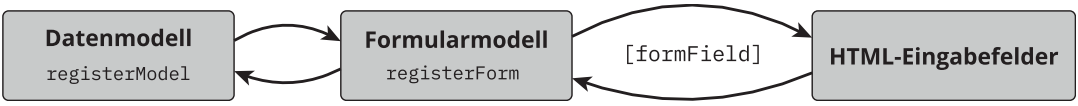


Abb. 25.2 *Datenfluss in Signal Forms*

Neben `value` bietet das Objekt `FieldState` noch weitere Signals an. Eine Auswahl haben wir in Tabelle 25.1 aufgelistet.

Bitte beachte, dass die Zustände `valid` und `invalid` keine exakten Gegenteile sind: `invalid` ignoriert die asynchrone Validierung mit dem Zustand `pending`, siehe dazu Unterabschnitt 25.5.6 ab Seite 315.

Eigenschaft	Typ	Beschreibung
`value`	`WritableSignal`	aktueller Wert
`validinvalid`	`Signal<boolean>`	Gültigkeitsstatus (inkl. Kindfelder)
`touched`	`Signal<boolean>`	Interaktionsstatus *(Wurde das Feld fokussiert und wieder verlassen?)*
`dirty`	`Signal<boolean>`	Änderungsstatus *(Wurde der Wert verändert?)*
`errors`	`Signal<ValidationError[]>`	Liste von Validierungsfehlern
`hidden`	`Signal<boolean>`	gibt an, ob das Feld als versteckt markiert ist
`readonly`	`Signal<boolean>`	gibt an, ob das Feld schreibgeschützt ist
`disabled`	`Signal<boolean>`	gibt an, ob das Feld deaktiviert ist
`disabledReasons`	`Signal<DisabledReason[]>`	Liste mit Gründen für die Deaktivierung
`submitting`	`Signal<boolean>`	gibt an, ob das Formular gerade abgeschickt wird
`pending`	`Signal<boolean>`	gibt an, ob asynchrone Validierungen ausstehen

Tab. 25.1
Reaktive Eigenschaften von FieldState

25.5 Schema für Validierung und Feldlogik

Zu einem guten Formular gehört auch eine Validierung der Eingabedaten. Das trägt zur Barrierefreiheit und Verbesserung der User Experience bei, indem wir direktes Feedback zu den Eingaben anzeigen. In Signal Forms definieren wir solche Regeln mithilfe eines Formularschemas.

Ein *Schema* ist eine Funktion, die Logik für einen Teil eines Formulars beschreibt. Das kann ein komplettes Formular sein, aber auch ein Teilbereich oder ein einzelnes Feld. Zur Erzeugung eines Schemas können wir die Funktion `schema<T>()` verwenden. Der generische Typparameter `T` gibt an, für welchen Teil der Formularstruktur das Schema verantwortlich ist.

Die übergebene Funktion erhält als Argument ein Objekt vom Typ `SchemaPathTree<T>`. Dieses Objekt spiegelt die Struktur des angegebenen Typs wider: Verwenden wir den Typ des Datenmodells (hier: `RegisterFormData`), repräsentiert der Schemapfad die gesamte Formularstruktur. Wir können durch dieses Objekt navigieren und einzelne Fel-

28 BookManager: Suche auf dem Server

Bisher haben wir die Suche in der Buchliste rein lokal ausgeführt. Das ist nicht optimal, denn wir laden immer alle Bücher, auch wenn wir nur einen Teil davon brauchen. Deshalb wollen wir die Suchfunktion umbauen und nur die Bücher vom Server anfordern, die zur Suchanfrage passen.

28.1 Praktische Umsetzung

Die Suche in der `BooksOverviewPage` soll auf dem Server ausgeführt werden. Immer wenn sich der Suchbegriff ändert, wollen wir eine neue HTTP-Anfrage stellen und nur die gefilterte Buchliste vom Server herunterladen. So vermeiden wir, dass die komplette Liste heruntergeladen werden muss, obwohl wir nur eine Auswahl von Büchern anzeigen möchten. Wir wollen dafür die Resource API einsetzen und die Abfragen mit einem Parameter steuern.

Außerdem wollen wir den eingegebenen Suchbegriff in der URL persistieren: Über einen Query-Parameter wird der Begriff an die URL angehängt, sodass wir eine bestimmte Suche bookmarken oder als Link versenden können.

Suche auf dem Server ausführen

Um die Liste der Bücher vom Server abzurufen, nutzen wir die Methode `getAll()` aus dem Service `BookStore`. Sie sendet einen HTTP-GET-Request an die BookManager API unter dem Pfad `/books`. Dabei werden standardmäßig alle Bücher aus der Datenbank abgerufen. Die Schnittstelle akzeptiert allerdings einen optionalen Query-Parameter `filter`, mit dem wir in den Feldern der Bücher suchen können.

Um die Suche vom Frontend an die API zu übergeben, müssen wir die Methode anpassen: Sie soll nun einen Suchbegriff entgegennehmen,

um damit die Suche durchzuführen. Wir wollen die Suche aber nicht nur einmalig ausführen, sondern auch bei Änderungen des Suchbegriffs soll die Buchliste stets neu abgerufen werden. Deshalb erwartet `getAll()` eine Funktion, die den aktuellen Suchbegriff zurückgibt.

Wir reichen diese Funktion an `httpResource()` weiter. Dort läuft sie in einem Reactive Context und reagiert auf Änderungen der darin verwendeten Signals. Die Resource sorgt auch automatisch dafür, dass bereits gestartete Requests abgebrochen werden, wenn sich der Suchbegriff erneut ändert, bevor der Server ein Resultat geliefert hat.

Aus der Funktion geben wir keinen einfachen String für die URL zurück, sondern ein Objekt mit weiteren Optionen. Es enthält neben der URL auch das Property `params`, in dem wir Query-Parameter übergeben können, die an den Pfad angehängt werden. Hier nutzen wir den Rückgabewert der Funktion `searchTerm()` als Wert für den Query-Parameter `filter`. Immer wenn sich der Suchbegriff ändert, wird ein neuer HTTP-Request an die URL `/books?filter=<Suchbegriff>` gestellt, und der Server liefert nur die Bücher zurück, die zu der Suche passen.

Listing 28.1 *Suchparameter an* httpResource() *übergeben* (book-store.ts)

```
// ...
@Injectable({ /* ... */ })
export class BookStore {
  // ...
  getAll(searchTerm: () => string): HttpResourceRef<Book[]> {
    return httpResource<Book[]>(
      () => ({
        url: `${this.#apiUrl}/books`,
        params: { filter: searchTerm() }
      }),
      { defaultValue: [] }
    );
  }
  // ...
}
```

In der Komponente `BooksOverviewPage` müssen wir nun etwas aufräumen: Wir entfernen das Computed Signal `filteredBooks`, denn die Filterung wird nun auf dem Server ausgeführt.

Der eingegebene Suchbegriff liegt immer aktuell im Signal `searchTerm` vor. Wir übergeben an die Methode `getAll()` eine Funktion, die das Signal aufruft. Diese Funktion wird später im Reactive Context ausgeführt. Die Resource reagiert damit auf Änderungen des Signals und führt den Request erneut aus, wenn sich der Suchbegriff ändert.

```
// ...
@Component({ /* ... */ })
export class BooksOverviewPage {
  #bookStore = inject(BookStore);
  // ...
  protected searchTerm = signal('');

  protected books = this.#bookStore
        ↪ .getAll(() => this.searchTerm());
  protected likedBooks = signal<Book[]>([]);

  protected filteredBooks = computed(() => {
    if (!this.books.hasValue()) {
      return [];
    }

    if (!this.searchTerm()) {
      return this.books.value();
    }

    const term = this.searchTerm().toLowerCase();
    return this.books.value().filter((b) => b.title
        ↪ .toLowerCase().includes(term));
  });
  // ...
}
```

Listing 28.2 *Die gefilterte Buchliste abrufen* (books-overview-page.ts)

Zum Abschluss müssen wir noch das Template der Komponente `BooksOverviewPage` anpassen. Hier iterieren wir jetzt nicht mehr über `filteredBooks`, sondern direkt über die Werte der Resource `books`, auf die wir mit `value()` zugreifen können. Zuvor müssen wir mit der Methode `hasValue()` prüfen, ob die Resource einen Wert besitzt.

```
<!-- ... -->
@if (books.hasValue()) {
  @for (b of books.value(); track b.isbn) {
    <app-book-card [book]="b" (like)="addLikedBook($event)" />
  }
}
```

Listing 28.3 *Die gefilterte Liste anzeigen* (books-overview-page.html)

Wir haben somit die Suchfunktion vom Frontend ins Backend verlagert. Betrachten wir die Netzwerkanfragen in den Developer Tools des Browsers, sehen wir, dass bei jeder Eingabe im Suchfeld ein HTTP-Request mit dem Query-Parameter `filter` an die API geschickt wird. Wenn wir schnell tippen, werden bereits gestartete Requests abgebrochen, wenn sie noch nicht abgeschlossen sind.

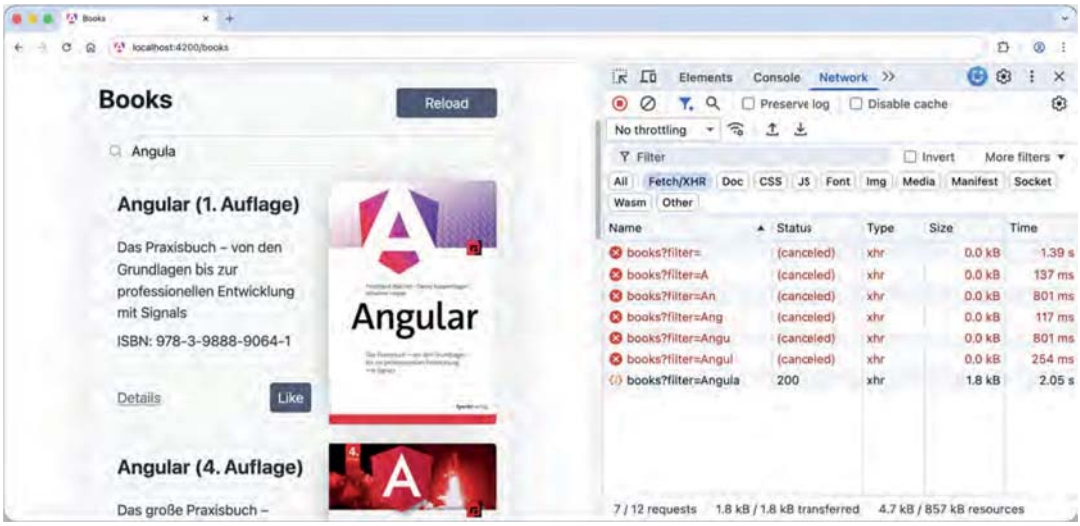


Abb. 28.1
Netzwerkanfragen für die Suche beobachten

Suchbegriff mit URL-Parameter synchronisieren

Der Suchbegriff befindet sich im Suchfeld in der Oberfläche. Laden wir die Seite in diesem Zustand neu, geht die Eingabe allerdings verloren!

Wir wollen den eingegebenen Suchbegriff deshalb in der Seiten-URL in einem Query-Parameter ablegen: `books?search=<Suchbegriff>`. So ist es möglich, die vorgefilterte Buchliste als Bookmark zu speichern oder den Link zu verschicken. Rufen wir eine URL mit Suchbegriff später auf, wollen wir die gleiche Suche mit dem Ergebnis noch einmal sehen.

Der Suchbegriff soll also stets zwischen dem Eingabefeld und der URL synchronisiert werden. Dafür müssen wir drei wesentliche Schritte erledigen:

1. Wir müssen einen Suchparameter in der URL auslesen.
2. Befindet sich ein Suchbegriff in der URL, müssen wir diesen ins Suchfeld übernehmen.
3. Ändern wir den Suchbegriff im Eingabefeld, müssen wir ihn in der URL aktualisieren, sodass Suchfeld und URL-Parameter stets synchron sind.

Um Verwechslungen zu vermeiden, möchten wir noch einmal auf die verschiedenen Pfade hinweisen, mit denen wir es in diesem Kapitel zu tun haben: Wir haben im vorherigen Abschnitt eine HTTP-Anfrage an die BookManager API gestellt, um die Bücher abzurufen, und haben den

Suchbegriff dabei im Query-Parameter `filter` an den Server übermittelt. In diesem Abschnitt geht es nun darum, den Suchbegriff in der URL der Angular-Anwendung abzulegen, die in der Adresszeile des Browsers steht. Dabei unterstützt uns der Router von Angular.

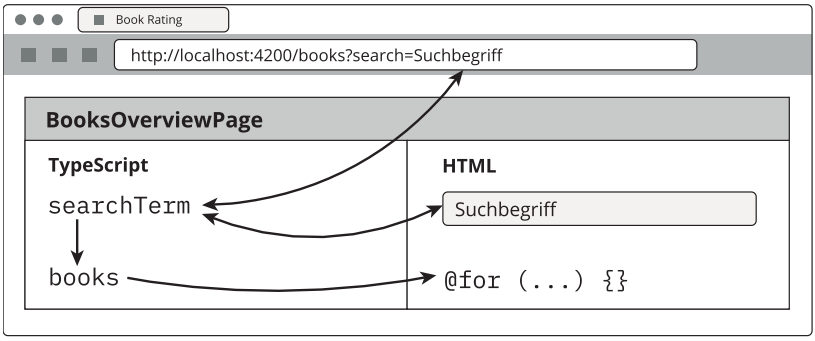


Abb. 28.2 *Suche: Kommunikation in der* BooksOverviewPage

Suchbegriff aus der URL lesen

Um den Query-Parameter in der Komponente zu empfangen, können wir das Component Input Binding verwenden, das wir in den Kapiteln 17 und 18 kennengelernt haben.

In der Komponente definieren wir ein Input Signal mit dem Namen `search`, das die Typen `string` oder `undefined` annehmen kann. Der gewünschte Typ `InputSignal<string | undefined>` ergibt sich automatisch, wenn wir keinen Startwert an die Funktion `input()` übergeben. Der Typ `undefined` ist hier notwendig, weil der Query-Parameter optional ist: Wird er nicht in der URL übergeben, setzt der Router das Input auf `undefined`.

Der zu suchende Begriff kann nun aus zwei Quellen stammen: aus dem Query-Parameter und aus dem Eingabefeld in der Komponente. Um diese beiden Quellen zusammenzuführen, können wir ein Linked Signal einsetzen: Es berechnet seinen Wert automatisch auf Basis einer Funktion (so wie ein Computed Signal), kann aber jederzeit manuell gesetzt werden. Wir können also den Wert des Query-Parameters verwenden und zusätzlich die Eingaben aus dem Suchfeld berücksichtigen. Das Property `searchTerm` initialisieren wir mit einem Linked Signal, das sich bei Änderungen im Input `search` aus dessen Wert berechnet. Damit hier stets ein String vorliegt, prüfen wir den Wert von `search` und geben alternativ einen leeren String zurück.

34 AI-Unterstützung für Angular

Softwareprojekte werden komplexer, und Anforderungen steigen. Werkzeuge für Artificial Intelligence (AI)[1] können uns bei der Entwicklung unterstützen und Entlastung schaffen: Sie helfen unter anderem beim Generieren von Code, sie erklären komplexe Zusammenhänge und sie schlagen Verbesserungen vor.

Den Weg in den Alltag fand AI durch browserbasierte Chats wie ChatGPT, Gemini oder Perplexity. Doch wer damit Software entwickelt, stößt schnell an Grenzen: Der Chat kennt das Projekt nicht, und Code muss manuell hin- und herkopiert werden. Einen Schritt weiter gehen *AI-Agenten*: Sie haben direkten Zugriff auf das Projekt, können Dateien lesen und bearbeiten, Befehle ausführen und mehrere Schritte autonom planen. Agenten können also prinzipiell alles tun, was wir am Computer auch tun könnten. Die Agenten laufen typischerweise in einer Sandbox und fragen bei kritischen Aktionen nach Bestätigung.

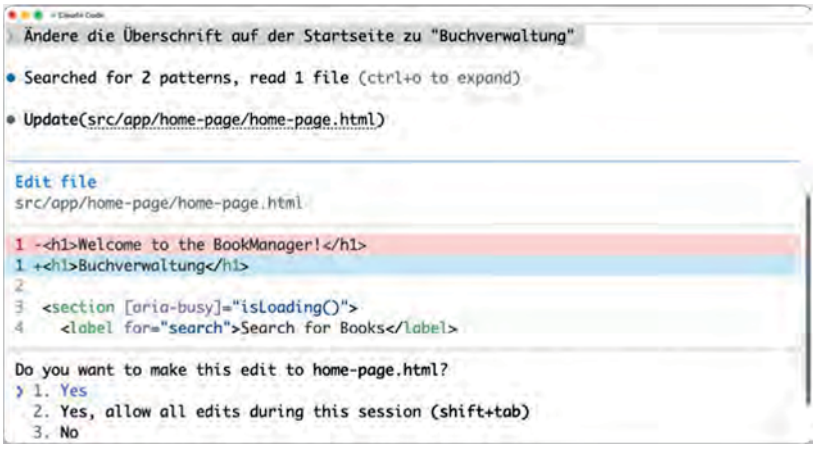


Abb. 34.1
Ein AI-Agent fragt vor einer Änderung nach Bestätigung.

[1] Genau genommen handelt es sich nicht um echte Intelligenz, sondern um statistische Mustererkennung auf Basis großer Textmengen. Die populären Begriffe »Künstliche Intelligenz« beziehungsweise »Artificial Intelligence« haben sich trotzdem im Sprachgebrauch etabliert, und wir verwenden sie hier ebenso.

Angular bietet für die Arbeit mit solchen Agenten spezielle Unterstützung, damit wir optimale Ergebnisse erhalten und der generierte Code den aktuellen Best Practices entspricht. Bevor wir ins Detail gehen, sollten wir aber besprechen, warum diese Unterstützung überhaupt notwendig ist.

34.1 Herausforderung: veraltetes Wissen

Die technische Grundlage aller AI-Agenten ist ein Large Language Model (LLM). Es basiert auf Trainingsdaten, die zu einem bestimmten Zeitpunkt erstellt wurden. Da ein solches Training extrem ressourcenintensiv ist, wird es nicht permanent durchgeführt. Es gibt also praktisch einen Stichtag, und selbst die besten Modelle können nur das »wissen«, was bis zu diesem Datum existierte.

Problematisch wird das bei schnelllebigen Technologien wie Angular: Neue Features kommen hinzu und Best Practices ändern sich. Aktuelle Neuerungen wie Signal Forms, die Resource API oder Angular Aria sind womöglich nicht in den Trainingsdaten vorhanden. Ältere Konzepte wie das Modulsystem (`NgModule`) oder die Strukturdirektiven (`NgIf` und `NgFor`) sind dagegen dem Modell bestens bekannt. Bedenkt man zudem, dass ältere Konzepte über Jahre hinweg mehr Dokumentation, Tutorials und Codebeispiele angesammelt haben, ist es statistisch wahrscheinlicher, dass das Modell diese vorschlägt. Für die Wartung bestehender Legacy-Projekte ist dies ein Vorteil. Wer aber eine moderne Anwendung mit aktuellen Best Practices anstrebt, erhält vom Modell mit höherer Wahrscheinlichkeit ältere Lösungsansätze. Im ungünstigsten Fall vermischt das Modell alte und neue Konzepte oder *halluziniert*, das heißt, es erzeugt plausibel klingende, aber faktisch falsche Aussagen. Das Ergebnis ist inkonsistenter oder nicht funktionierender Code.

Die Lösung liegt darin, dem AI-Agenten den notwendigen Kontext bereitzustellen. Angular bietet dafür mehrere Werkzeuge:

- **Konfigurationsdateien** für Instruktionen und Beispiele
- **MCP-Server** für Angular-spezifische Informationen (und Tools)

Angular unterstützt außerdem *Agent Skills*, also Anleitungen für AI-Agenten in Form einer Datei `SKILL.md`. Unter *https://github.com/angular/skills* stehen mit `angular-new-app` und `angular-developer` zwei offizielle Skills bereit.

34.2 AI-Konfigurationsdateien

Zu Beginn ihrer Arbeit benötigen AI-Agenten möglichst viele gute Informationen. Man spricht hier auch von Kontext. Der Hersteller hat bereits grundlegende Regeln und Instruktionen bereitgestellt, den sogenannten *System Prompt*. Doch das reicht in der Regel nicht aus: Der Agent hat noch keine Kenntnis über das spezifische Projekt, für das er Arbeit erledigen soll.

Hier kommen projektspezifische Konfigurationsdateien ins Spiel. Die meisten AI-Agenten suchen nach solchen Dateien mit einem bestimmten Namen: Claude Code erwartet `.claude/CLAUDE.md`, Cursor verwendet `.cursorrules`, GitHub Copilot nutzt `.github/copilot-instructions.md` und so weiter. Jeder Hersteller hat seinen eigenen Standard, doch der generische Dateiname `AGENTS.md` könnte sich als herstellerübergreifender Standard etablieren. Diese Dateien enthalten eine Sammlung von Regeln und Best Practices für das jeweilige Projekt. Man spricht hier von einem *Custom Prompt*: eine Datei mit projektspezifischen Anweisungen, die das Verhalten des AI-Agenten steuert und den System Prompt ergänzt. Man könnte diese Eingaben auch vor jeder Konversation manuell eingeben, aber das wäre aufwendig, und man vergisst es schnell.

Da sich noch kein einheitlicher Standard für den Dateinamen etabliert hat, unterstützt die Angular CLI verschiedene Varianten. Sie fragt beim Erzeugen einer Anwendung nach dem eingesetzten Agenten und generiert die passenden Dateien. Wir können die Konfiguration auch direkt mit der Option `--ai-config` angeben:

```
$ ng new my-app --ai-config=agents
```

Haben wir uns zu Beginn gegen eine explizite Konfiguration entschieden oder wollen eine weitere ergänzen, so können wir nachträglich eine solche Konfiguration erzeugen:

```
$ ng g ai-config
```

Die Richtlinien umfassen Best Practices für TypeScript und Angular, Vorgaben für Komponenten, State Management, Templates und Services sowie Anforderungen an Barrierefreiheit.

Der Agent hat nun Instruktionen. Doch ob er diese korrekt umsetzen kann, hängt von seinem Wissensstand ab. Im Custom Prompt steht etwa, dass die neue Syntax für den Control Flow verwendet werden soll.

Aber woher soll das Modell wissen, wie diese funktioniert, wenn sie zum Zeitpunkt des Trainings noch nicht existierte?

Hier können wir teilweise nachhelfen: Die von Angular generierte Datei ist ein guter Startpunkt, aber wir können sie erweitern. Nützlich sind konkrete Beispiele für neue Syntax, projektspezifische Konventionen oder Hinweise zu Fehlern, die der Agent wiederholt gemacht hat. Für den Custom Prompt gilt: kurz und fokussiert halten. Zu viele Instruktionen verwässern die Wirkung. Der AI-Agent kann übrigens selbst beim Formulieren guter Instruktionen helfen. Auch der MCP-Server, den wir später vorstellen, kann fehlende Informationen bereitstellen.

Allerdings gibt es eine Einschränkung: Jedes LLM kann nur eine begrenzte Menge an Text gleichzeitig verarbeiten. Man spricht von einem Kontextfenster. Der Custom Prompt liegt in diesem Fenster, und bei längeren Sessions können die Instruktionen in Vergessenheit geraten.

34.3 Herausforderung: das Kontextfenster

Wird das Kontextfenster überschritten, »vergisst« der AI-Agent frühere Teile der Konversation. Dieses Vergessen ist technisch notwendig, damit die Unterhaltung weitergehen kann. Das häufigste Mittel besteht darin, die bisherige Konversation bestmöglich zusammenzufassen (engl. *Context Summarization*). Das funktioniert manchmal hervorragend und manchmal leider überhaupt nicht. Hat die Zusammenfassung wichtige Aspekte entfernt, führt dies zu inkonsistenten Antworten oder veralteten Codevorschlägen.

Damit verwandt ist der *Lost-in-the-Middle*-Effekt: Informationen, die in der Mitte eines sehr langen Kontexts stehen, werden vom Modell weniger stark berücksichtigt als Informationen am Anfang oder Ende. Das kann dazu führen, dass initiale Instruktionen aus den Custom Prompts im Laufe der Konversation vernachlässigt werden und das Modell nur noch auf den ursprünglichen System Prompt zurückfällt. Je länger die Session andauert, desto wahrscheinlicher werden solche Effekte. Moderne AI-Agenten nutzen neben der Zusammenfassung weitere Strategien, z. B. gezielte Tool-Aufrufe oder Sub-Agenten mit eigenem Kontextfenster. Eine besonders elegante Lösung bietet der MCP-Server der Angular CLI.

35 Softwaretests

Softwaretests sind aus professioneller Softwareentwicklung nicht wegzudenken. Sie decken Fehler früh auf, erzwingen eine saubere Architektur und schützen bestehenden Code vor Regressionen. So bleibt mehr Zeit für Features und weniger für Fehlerjagd. In diesem Kapitel lernst du die Grundlagen von Testing im Allgemeinen und das Tooling von Angular im Speziellen kennen.

35.1 Testarten: Wie sollte man testen?

Je nachdem, wie viel Code ein Test abdeckt und wie realitätsnah die Testumgebung ist, unterscheiden wir verschiedene Testarten. Ein Test kann eine einzelne Funktion isoliert prüfen oder eine komplette Anwendung im echten Browser durchspielen. Zwischen diesen Extremen liegt ein breites Spektrum.

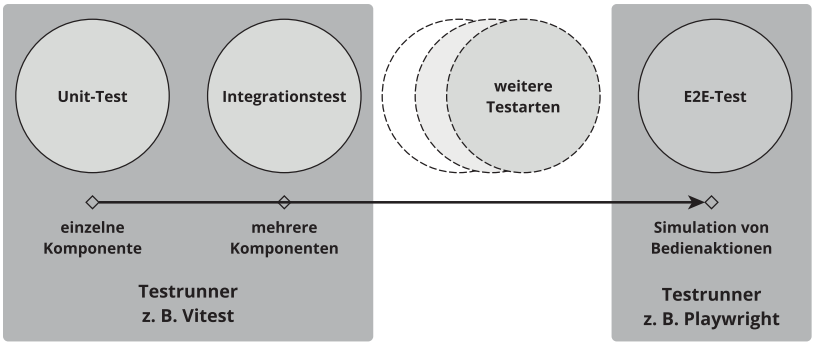


Abb. 35.1
Arten von Tests

Für die Arbeit mit Angular sind vor allem drei Arten von Tests relevant:

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

Unit-Tests überprüfen die kleinsten Einheiten (Units) einer Software. In Angular sind das typischerweise Services, Pipes und Komponenten, die wir im praktischen Teil ab Kapitel 5 vorstellen. Jeder andere Code, der nicht Teil der gerade getesteten Einheit ist, wird als *Abhängigkeit* bezeichnet. Bei einem Unit-Test ist es wichtig, dass wirklich nur eine einzige Einheit getestet wird. Das bedeutet, dass wir alle Abhängigkeiten durch sogenannte *Stubs* bzw. *Mocks* ersetzen sollten.

Bei vielen Abhängigkeiten kann es aufwendig sein, alle zu ersetzen. Es ist dann effektiver, mehr als nur eine Einheit mit einem Test abzudecken. Sind mehrere Softwareeinheiten im Test involviert, so nennt man dies einen *Integrationstest*. Integrationstests können auch gezielt das Zusammenspiel von bereits einzeln getesteten Units absichern. In Angular verwischen die Grenzen zwischen Unit- und Integrationstests oft. Das liegt daran, dass Komponenten ihre Kindkomponenten (also Komponenten, die im Template eingebunden sind) automatisch mitbringen und standardmäßig auch die echten Services geladen werden. Das ist aber nicht schlimm, solange wir gezielt steuern, was wir testen, und externe Abhängigkeiten bei Bedarf durch Mocks ersetzen.

Neben Unit- und Integrationstests gibt es weitere Testarten, die je nach Kontext und Tooling unterschiedlich benannt werden. Tests auf höherer Ebene sind weitgehend unabhängig vom eingesetzten Framework, weshalb wir sie in diesem Buch nur kurz anreißen wollen.

E2E-Tests (auch Oberflächentests genannt) ergänzen unsere Unit- und Integrationstests um die Perspektive der Nutzenden unserer Anwendung. Der Name »Ende zu Ende« beschreibt dabei den Testumfang: Ein Ende ist immer der Browser. Das andere Ende ist nicht eindeutig definiert und Teil der persönlichen Teststrategie: Es könnte das Backend mit echter Datenbank sein, ein Backend mit fest einprogrammierten Testdaten oder auch nur das Frontend mit gemockten Schnittstellen. Ein echter Browser wie Chrome oder Firefox wird ferngesteuert und testet die vollständige Anwendung. Während Unit- und Integrationstests einzelne technische Anforderungen prüfen, spezifizieren E2E-Tests komplette fachliche Abläufe.

Wir werden uns in diesem Buch mit Unit- und Integrationstests beschäftigen. Für E2E-Tests nutzt man eigenständige Frameworks, die unabhängig von Angular arbeiten. Es existieren aber Integrationen für die Angular CLI, sodass wir die Tests bequem mit dem Befehl `ng e2e` starten können. Mehr dazu erfährst du am Ende dieses Kapitels in Abschnitt 35.4 ab Seite 505.

35.2 Unit- und Integrationstests mit Vitest

Ein Angular-Projekt bringt von Haus aus alles mit, um Unit- und Integrationstests zu schreiben. Als Testrunner kommt Vitest zum Einsatz.[1] Ein Testrunner findet unsere Tests, führt sie aus und meldet die Ergebnisse. Vitest ist ein modernes, schnelles Testwerkzeug für JavaScript und TypeScript und ist bereits eingerichtet, sodass wir direkt loslegen können.

Alle Testdateien für Vitest tragen das Suffix `.spec.ts`. Jede Testdatei befindet sich idealerweise im selben Verzeichnis wie die zu testende Datei und hat denselben Namen. Zur Komponente `HomePage` mit dem Dateinamen `home-page.ts` sollte es also die Datei `home-page.spec.ts` geben. So finden wir den dazugehörigen Test sofort.

35.2.1 Tests starten

Um alle Tests des Projekts auszuführen, nutzen wir folgenden Befehl:

```
$ ng test
```

Listing 35.1
Tests starten

Bei vielen Tests kann es hilfreich sein, nur einen Teil davon auszuführen. Mit `--include` können wir nur Tests aus Dateien ausführen, die zu einem Glob-Pattern passen. Ein Glob-Pattern ist ein Suchmuster mit Wildcards: `*` steht für beliebige Zeichen, `**` für beliebige Verzeichnistiefe. Mit `--filter` lassen sich einzelne Testfälle anhand ihres Titels auswählen:

```
# Tests im Verzeichnis "feature-a" ausführen
$ ng test --include **/feature-a/**/*.spec.ts
# Tests ausführen, die "should create" im Titel haben
$ ng test --filter 'should create'
```

Listing 35.2
Ausgewählte Tests starten

Vitest führt unsere Tests standardmäßig nicht in einem echten Browser aus, sondern mit Node.js. Dabei kommt die Bibliothek jsdom[2] zum Einsatz, die den Browser emuliert. Die Testergebnisse können wir somit direkt in der Konsole betrachten.

[1] *https://ng-buch.de/d/vitest* – Vitest

[2] *https://ng-buch.de/d/jsdom* – GitHub: jsdom

Limitierungen von jsdom

jsdom ist eine Browser-Emulation und kann nicht alles nachbilden. Features wie echte Browser-APIs (z. B. `IntersectionObserver`, `ResizeObserver`) sind in jsdom nicht implementiert und müssen per Mock oder Polyfill nachgerüstet werden. CSS-Layout-Berechnungen funktionieren ebenfalls nur eingeschränkt. Für solche Fälle können Tests auch in einem echten Browser ausgeführt werden.[a]

[a] *https://ng-buch.de/d/vitest-browser-mode* – Vitest: Browser Mode

Bei Änderungen wird der Code automatisch erneut kompiliert und die Tests werden neu ausgeführt: Dies ist der Watch-Modus. Wir können den Prozess beenden, indem wir die Taste q drücken.

Wenn die Tests nur einmal ausgeführt werden sollen, können wir den Watch-Modus wie folgt deaktivieren:

Listing 35.3 *Tests einmalig ausführen*

```
$ ng test --watch=false
```

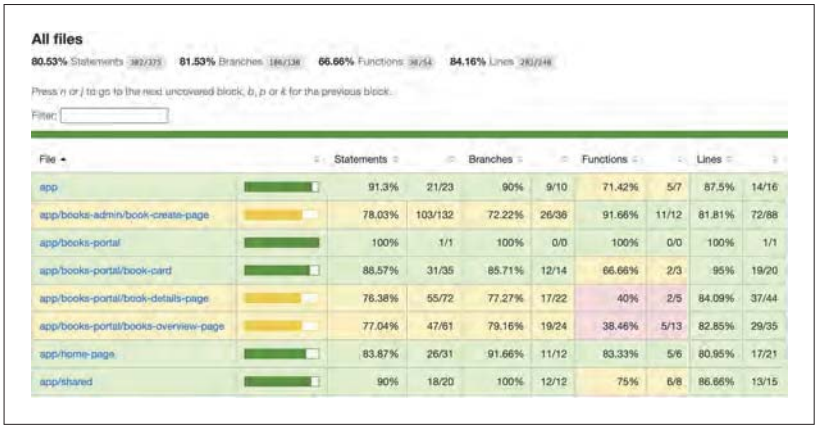
Um die Testabdeckung (Code Coverage) zu messen, können wir folgenden Befehl nutzen:

Listing 35.4 *Testabdeckung prüfen*

```
$ ng test --coverage
```

Vitest erstellt dann einen Coverage Report im Ordner `coverage`. Die darin enthaltene `index.html` können wir mit einem Browser unserer Wahl öffnen.

Abb. 35.2 *Coverage Report*



35.2.2 Der Aufbau eines Tests

Mit einem Test beweisen wir, dass unsere Software funktioniert, wie sie soll. Ein solcher Beweis folgt immer demselben Aufbau: Wir bereiten eine Situation vor, führen eine Aktion aus und prüfen das Ergebnis. Dieses Muster trägt den Namen *AAA-Pattern*, kurz für *Arrange, Act, Assert*. Die konkrete Ausgestaltung variiert je nach Programmiersprache und Tooling, aber im Kern gehen wir immer inhaltlich gleich vor:

- **Arrange:** Wir bereiten die Testumgebung vor und erstellen alle benötigten Objekte.
- **Act:** Wir führen die zu testende Aktion aus.
- **Assert:** Wir überprüfen, ob das Ergebnis unseren Erwartungen entspricht.

Das AAA-Pattern ist zwar universell, aber die konkrete Syntax unterscheidet sich je nach Tooling. In den frühen 2000ern wurde durch die Ruby-Community ein Stil populär, der Tests als lesbaren Text formuliert: `describe`, `it`, `should`. Dieser Stil wurde in der JavaScript-Community übernommen und hat sich seitdem als Standard etabliert. Das folgende Beispiel erbringt den trivialen Beweis, dass 2 + 3 = 5 ist:

Listing 35.5
Das AAA-Pattern anwenden

```
function add(a: number, b: number) {
  return a + b;
}

describe('add function', () => {
  let testNumbers: number[];

  beforeEach(() => {
    // Arrange: Testdaten vorbereiten
    testNumbers = [2, 3];
  });

  it('should add two numbers', () => {
    // Arrange
    const [a, b] = testNumbers;

    // Act
    const result = add(a, b);

    // Assert
    expect(result).toBe(5);
  });
});
```