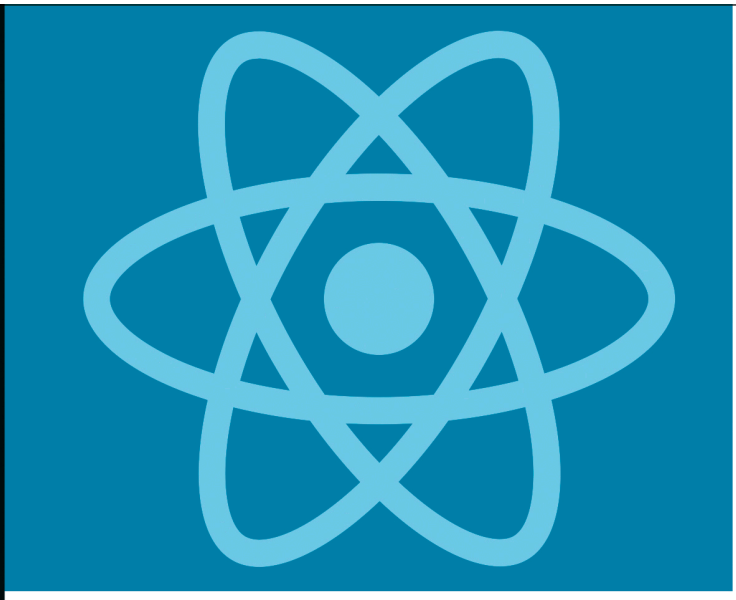


Inkl. React Native, Redux und Next.js

SPA, Props, Component, Hooks, SSR, TypeScript, Flow, PWA

```
export function createContext<T>(
  defaultValue: T,
  calculateChangedBits: ?(a: T, b: T) => number
): ReactContext<T> {
  if (calculateChangedBits === undefined) {
    calculateChangedBits = null;
  } else {
    if (__DEV__) {
      warningWithoutStack(
        calculateChangedBits === null ||
          typeof calculateChangedBits === 'function',
        'createContext: Expected the optional second
        argume
          'funct
```

Sebastian Springer

React

Das umfassende Handbuch

- Für Einsteiger und Fortgeschrittene
- Komponentenarchitektur, Hooks, State-Management und Formulargestaltung
- Tests, Performance, Routing, Server Side Rendering mit Next.js, Authentifizierung, KI-Applikationen u. v. m.

3., aktualisierte und erweiterte Auflage

Alle Beispielprojekte zum Download

Rheinwerk Computing

Kapitel 4

Typsicherheit in React-Applikationen mit TypeScript

In diesem Kapitel erfahren Sie, wie TypeScript Sie dabei unterstützt, klarer strukturierten Code zu schreiben und typische Fehler bereits während der Entwicklung zu vermeiden.

JavaScript verfügt über ein schwaches Typsystem, das lediglich eine Handvoll Typen unterstützt. Es ist nicht möglich, Variablen einen festen Typ zuzuordnen, genauso wenig, wie Sie die Signatur von Funktionen mit Typen versehen können. Je umfangreicher eine Applikation wird und je mehr Entwickler beteiligt sind, desto mehr macht sich das Fehlen eines strikten Typsystems in JavaScript bemerkbar.

Um dieses Problem zu beheben, gibt es Typsysteme, die auf JavaScript basieren. In diesem Kapitel lernen Sie mit *TypeScript* den De-facto-Standard kennen, wenn es um typsicheres JavaScript geht, und sehen, wie es mit React funktioniert.

4.1 Was bringt ein Typsystem?

Allgemein gesprochen, erhalten Sie durch ein Typsystem zusätzliche Struktur und Sicherheit. Durch die Festlegung, welchen Typ eine bestimmte Variable, ein Parameter oder der Rückgabewert einer Funktion hat, schränken Sie zwar die Freiheit ein, die JavaScript Ihnen bietet. Es bedeutet aber auch, dass Sie nicht mehr versehentlich aus einer Stringvariablen eine Zahl oder ein Boolean machen können.

Wenn Sie sich strikt an die Vorgaben des Typsystems halten und bei jeder Variablen und Funktion die Typen angeben, zwingt Sie das auch dazu, sich mehr Gedanken über den Aufbau Ihrer Applikation zu machen. Damit geht einher, dass Sie Ihren Code dokumentieren. Ein Kommentarblock einer Funktion kann leicht veralten, da Sie nicht gezwungen werden, ihn bei Änderungen am Quellcode anzupassen. Bei der Verwendung eines Typsystems müssen Sie die Typangabe in der Signatur einer Funktion anpassen, da Sie ansonsten Fehlermeldungen bei der Überprüfung erhalten.

Die Lesbarkeit des Quellcodes wird durch den Einsatz eines Typsystems positiv beeinflusst. Bei einem Blick auf die Signatur einer Funktion sehen Sie auf einen Blick, was diese als Eingabe erwartet und was sie zurückgibt. Kombinieren Sie die Typen nun noch mit einer sprechenden Benennung der Funktion selbst sowie der Parameter, sollte auch je-

mand, der die Funktion nicht selbst geschrieben hat, auf einen Blick erkennen können, was sie tut.

Auch bei der Suche nach Fehlern und bei der Wartung von Applikationen kann ein Typsystem gute Dienste leisten, da es das Risiko vermindert, dass bei komplexen Abhängigkeiten Probleme entstehen, weil sie direkt im Quellcode festgehalten werden.

Der offensichtlichste Vorteil eines Typsystems ist eine verbesserte Unterstützung durch Programmierwerkzeuge wie die Entwicklungsumgebung oder durch Werkzeuge zur statischen Codeanalyse. Die meisten Entwicklungsumgebungen unterstützen die gängigen Typsysteme standardmäßig oder verfügen über Erweiterungen, die sich in wenigen Schritten installieren lassen. Ist Ihre Entwicklungsumgebung korrekt konfiguriert, erhalten Sie schon während der Entwicklung sofortiges Feedback zu Ihrem Quellcode. Dies geschieht zum einen über Fehlermeldungen, wenn Sie gegen die Regeln des Typsystems verstoßen, und zum anderen durch Autovervollständigung während der Entwicklung. Die Autovervollständigung liefert Ihnen mögliche Vorschläge, sobald Sie beispielsweise beginnen, den Namen einer Methode eines Objekts zu schreiben. Dieses Feature existiert zwar auch für natives JavaScript, ist jedoch bei der typsicheren Variante erheblich besser und zuverlässiger.

4.2 Die verschiedenen Typsysteme

Die Typsysteme für JavaScript lassen sich grob in zwei Kategorien unterteilen: in Werkzeuge, die mithilfe bestimmter Annotationen die Einhaltung der Regeln überprüfen, und in vollwertige Typsysteme, bei denen der Quellcode in einer eigenen Sprache formuliert wird. Bei der ersten Kategorie liegt der Quellcode bereits in JavaScript vor und wird lediglich um die Typangaben ergänzt. Bevor der Quellcode ausgeführt werden kann, müssen die Typangaben entfernt werden, da ansonsten Syntaxfehler geworfen werden würden. Ein typischer Vertreter dieser Art ist *Flow*.

Flow

Flow wurde im Jahr 2014 von Facebook veröffentlicht und wird seitdem als Open-Source-Projekt auf GitHub verwaltet. Die Website von Flow finden Sie unter *https://flow.org/*. Mittlerweile wird Flow kaum noch in größeren Projekten eingesetzt. Es gibt jedoch eine populäre Ausnahme: React. Flow ist das Typsystem hinter React. Wenn Sie einen Blick in den Quellcode von React werfen, finden Sie in nahezu jeder Datei einen Kommentarblock mit der Zeichenkette `@flow`. Mit dieser Annotation aktivieren Sie die Typüberprüfung für die aktuelle Datei.

Die Syntax von Flow ähnelt der von TypeScript. Sie können die Typen von Variablen und die Signaturen von Funktionen und Methoden festlegen und Typkonstrukte wie Generics nutzen.

Die zweite Art der Typsysteme, zu denen beispielsweise *TypeScript* gehört, nutzt ebenfalls Annotationen, um die Einhaltung der Typenregeln zu überprüfen. Der TypeScript-Code wird vor der Ausführung in JavaScript übersetzt. Dabei werden die Typangaben entfernt und zusätzlich bestimmte Features emuliert. Der TypeScript-Compiler ist in der Lage, verschiedene JavaScript-Versionen zu erzeugen – sowohl modernen Code, wie er nur in einem modernen Browser lauffähig ist, als auch älteren JavaScript-Quellcode, der der Spezifikation von ECMAScript 3 oder ECMAScript 5 folgt.

In diesem Buch liegt der Schwerpunkt auf TypeScript, da es in den meisten Projekten zum Einsatz kommt.

4.3 TypeScript in einer React-Applikation einsetzen

TypeScript wird seit 2012 von Microsoft als Open-Source-Projekt entwickelt. TypeScript ist eine eigene Programmiersprache, die den Kern von JavaScript erweitert und um zusätzliche Features ergänzt.

Grundsätzlich ist gültiger JavaScript-Code auch gültiger TypeScript-Code. In die andere Richtung ist die Aussage aber nicht wahr. Wenn Sie versuchen, TypeScript direkt im Browser auszuführen, führt dies in der Regel zu Syntaxfehlern und einem Abbruch der Applikation. TypeScript hat sich in den vergangenen Jahren als der De-facto-Standard für typsicheres JavaScript durchgesetzt.

Auf jeden Fall sollten Sie den Einsatz eines Typsystems für Ihre Applikation in Betracht ziehen, da die Vorteile die Nachteile klar überwiegen. Als Typsystem empfehle ich Ihnen ganz klar TypeScript, da es eine sehr hohe Verbreitung in der Community hat und React und sein gesamtes Ökosystem TypeScript hervorragend unterstützt.

4.3.1 TypeScript in eine React-Applikation einbinden

Die Kombination aus React und TypeScript hat sich seit längerer Zeit etabliert. Mit dem *Vite*-Template für React mit TypeScript haben Sie einen guten Startpunkt. Aber auch andere Projekte wie *Next.js* bieten Ihnen standardmäßig ein TypeScript-Setup an. In Listing 4.1 finden Sie den Befehl, den Sie zur Initialisierung der Applikation mit TypeScript verwenden:

```
npm create vite@latest my-app -- --template react-ts --no-interactive
```

Listing 4.1: Initialisierung einer Applikation mit TypeScript

Mit der Option `--template react-ts` sorgen Sie dafür, dass Vite alle Abhängigkeiten vorbereitet, die für die Verwendung von TypeScript in Ihrer Applikation erforderlich sind. Außerdem werden die benötigen Strukturen und Konfigurationen erstellt, sodass Sie direkt mit der Entwicklung beginnen können. Auch der Entwicklungs- und Build-Prozess wird entsprechend modifiziert.

Der erste Unterschied zu einer gewöhnlichen React-Applikation sind die Konfigurationsdateien für TypeScript im Wurzelverzeichnis. Konkret sind das die Dateien *tsconfig.app.json*, *tsconfig.node.json* und *tsconfig.json*. Die *tsconfig.json* ist die Hauptdatei, die die beiden anderen einbindet. Die *tsconfig.app.json* gilt nur für das *src*-Verzeichnis und generiert JavaScript-Code für den Browser. Die *tsconfig.node.json* kümmert sich um die Vite-Konfiguration und produziert Code für Node.js.

Ein weiterer Unterschied im Dateisystem ist, dass die Komponentendateien die Endung *.tsx* aufweisen, was andeuten soll, dass es sich um eine Kombination aus TypeScript und JSX handelt. Für Hilfsdateien, die kein JSX enthalten, nutzen Sie die Endung *.ts*.

Als erste TypeScript-Komponente setzen Sie die App-Komponente so wie in Listing 4.2 um:

```
import React from 'react';
import './App.css';

const App: React.FC = () => {
  let name: string = 'World';

  name = 42;
  return (
    <div className="App">
      <h1>Hello {name}</h1>
    </div>
  );
}

export default App;
```

Listing 4.2: TypeScript-Quellcode (»src/App.tsx«)

Bei TypeScript können Sie auf die explizite Angabe des Typs verzichten; die Angabe von `string` bei der `name`-Variablen ist also optional. Die erste Zuweisung eines Werts an eine Variable legt gleichzeitig ihren Typ fest. Die Zuweisung einer Zahl an die `name`-Variable führt zu einem Fehler, da eine String-Variable keine Zahl enthalten darf.

Ausführung in der Entwicklungsumgebung

TypeScript wird von allen gängigen Entwicklungsumgebungen unterstützt, beispielsweise von *WebStorm* oder *Visual Studio Code*. Der Quellcode wird unmittelbar bei seiner Erstellung überprüft, und Fehler werden sofort angezeigt. Mit diesem direkten Feedback lassen sich zahlreiche Fehler bereits vor der Ausführung des Quellcodes finden und beheben. Auch die Vorschläge für die Autovervollständigung sind erheblich besser als bei der Verwendung von reinem JavaScript, da die Signatur von Funktionen sowie die Struktur

von Objekten bekannt sind. In Abbildung 4.1 sehen Sie die Fehlermeldung, die Sie für den Quellcode aus Listing 4.2 erhalten, wenn Sie ihn in Visual Studio Code öffnen.

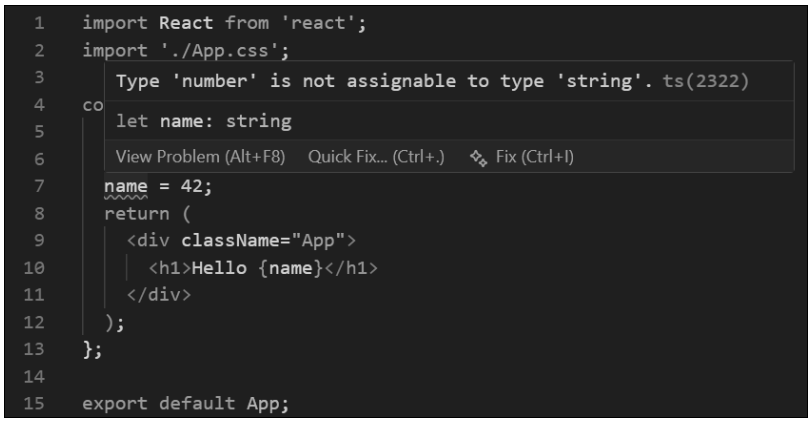


Abbildung 4.1: TypeScript-Fehlermeldung in Visual Studio Code

TypeScript lässt sich nicht nur in der Entwicklungsumgebung verwenden, sondern auch auf der Kommandozeile und kann so beispielsweise in einen automatisierten Build-Prozess eingebunden werden.

Ausführung auf der Kommandozeile

Das Herzstück von TypeScript ist der *TypeScript-Compiler*, kurz `tsc`. Dieses Kommandozeilenwerkzeug überprüft den TypeScript-Quellcode Ihrer Applikation und transformiert ihn in validen JavaScript-Quellcode, der im Browser lauffähig ist. Damit diese Transformation funktionieren kann, muss das `typescript`-Paket auf Ihrem System installiert sein. Vite übernimmt die Installation und Konfiguration für Sie.

Wenn Sie Ihre Applikation selbst aufbauen, erzeugen Sie mit dem Kommando `tsc --init` eine *tsconfig.json*-Datei. Die Konfigurationsdatei beeinflusst die Arbeitsweise des TypeScript-Compilers. Wechseln Sie auf die Kommandozeile und geben Sie das Kommando `npx tsc -b` ein, baut der TypeScript-Compiler das Projekt und überprüft dabei Ihren Code. Der Compiler findet automatisch die Konfigurationsdatei und wendet sie an. In Listing 4.3 sehen Sie die Ausgabe auf der Kommandozeile:

```
$ npx tsc -b
src/App.tsx:7:3 - error TS2322: Type 'number' is not assignable to
type 'string'.

7   name = 42;
    ~~~~

Found 1 error.
```

Listing 4.3: Ausgabe des TypeScript-Compilers auf der Kommandozeile

Die aktuelle Konfiguration von TypeScript sorgt dafür, dass während des Entwicklungsprozesses kein JavaScript-Quellcode im Dateisystem gespeichert wird. Dies ist nicht erforderlich, da TypeScript sehr tief in den Entwicklungsprozess integriert ist. Mit `npm run dev` führen Sie die Applikation direkt mit dem Vite-Dev-Server aus. Für den Produktivbetrieb müssen Sie den Quellcode in JavaScript übersetzen lassen. Dabei hilft Ihnen der Befehl `npm run build`. Er erzeugt eine lauffähige Applikation für Sie, die Sie auf einen beliebigen Webserver deployen können. Mit diesem Wissen können Sie nun tiefer in die Welt von TypeScript eintauchen.

4.3.2 Konfiguration von TypeScript

Sie können das Verhalten des TypeScript-Compilers durch Optionen auf der Kommandozeile oder über die *tsconfig.json*-Datei beeinflussen.

Die wichtigsten Angaben in der Konfiguration sind die `compilerOptions`, mit denen Sie den Compiler steuern können, und `include`, mit der Sie eine Liste von Verzeichnissen angeben können, die die Dateien mit dem TypeScript-Quellcode enthalten. Bei Bedarf können Sie mit dem `exclude`-Schlüssel bestimmte Muster vom Kompilierungsprozess ausschließen und mit `files` einzelne Dateien angeben. In Listing 4.4 sehen Sie die TypeScript-Konfiguration für Ihre Applikation, die Vite mit dem React-TypeScript-Template für Sie erzeugt:

```
{
  "compilerOptions": {
    "tsBuildInfoFile": "./node_modules/.tmp/tsconfig.app.tsbuildinfo",
    "target": "ES2022",
    "useDefineForClassFields": true,
    "lib": ["ES2022", "DOM", "DOM.Iterable"],
    "module": "ESNext",
    "types": ["vite/client"],
    "skipLibCheck": true,

    /* Bundler mode */
    "moduleResolution": "bundler",
    "allowImportingTsExtensions": true,
    "verbatimModuleSyntax": true,
    "moduleDetection": "force",
    "noEmit": true,
    "jsx": "react-jsx",

    /* Linting */
    "strict": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
```

```
    "erasableSyntaxOnly": true,
    "noFallthroughCasesInSwitch": true,
    "noUncheckedSideEffectImports": true
  },
  "include": ["src"]
}
```

Listing 4.4: TypeScript-Konfiguration einer React-Applikation (»tsconfig.app.json«)

In der Regel sollte die Standardkonfiguration für die Arbeit mit React ausreichen, sodass Sie mit der Konfiguration während des Entwicklungsprozesses kaum in Berührung kommen.

4.3.3 Die wichtigsten Features von TypeScript

TypeScript unterstützt primitive Datentypen wie `boolean`, `string` oder `number` sowie zusammengesetzte Datentypen wie Objekte und Arrays. Zudem gibt es spezielle Datentypen, wie beispielsweise `enum`, mit dem Sie Schlüssel auf Werte mappen können.

TypeScript erlaubt die Verwendung von Klassen, die gleichzeitig Typen darstellen und so in Typzuweisungen verwendet werden können. Darüber hinaus gibt es *Type Aliases*, *Interfaces* und *Generics*, die in React-Applikationen verwendet werden. In TypeScript können Sie mit *Modulen* arbeiten, also in sich geschlossenen Einheiten, die jeweils in einer Datei liegen. Die Modulsyntax entspricht der des ECMAScript-Modulsystems, sodass Sie in Ihrer Applikation weiterhin auf die Schlüsselwörter `import` und `export` zurückgreifen können, um Ihre Module zu importieren beziehungsweise zu exportieren.

Diese einzelnen Features lernen Sie im Laufe dieses Kapitels und im weiteren Verlauf dieses Buchs noch im Detail kennen.

4.3.4 Typdefinitionen – Informationen über Drittanbieter-Software

TypeScript weist eine Besonderheit im Umgang mit Bibliotheken auf, die in JavaScript verfasst sind und in einer TypeScript-Applikation verwendet werden sollen. Die Schnittstellen dieser Bibliotheken sind in der Regel nicht mit Typinformationen versehen, sodass Sie an dieser Stelle die meisten Vorteile von TypeScript verlieren würden, da der Compiler auf den impliziten `any`-Typ zurückfallen würde. Verwenden Sie `any`, ist eine weitere Typüberprüfung an dieser Stelle nicht mehr möglich.

Um dieses Problem zu lösen, stellt TypeScript mit Typdefinitionen ein Feature zur Verfügung, mit dem sich JavaScript-Schnittstellen mit Typen versehen lassen. Immer mehr Bibliotheken im React-Umfeld liefern diese Typdefinitionen gleich mit dem eigentlichen Quellcode aus, sodass die Versionen der Schnittstelle und der Typdefinition identisch sind. Weit häufiger werden die Typdefinitionen jedoch als zusätzliches Paket angeboten. Hierbei entsteht die Gefahr, dass die Versionen unterschiedlich sind und Sie mit einer po-

tenziell veralteten Version der Schnittstellendefinition arbeiten. Oft werden die Typdefinitionen jedoch auch von den Personen gepflegt, die die Bibliotheken erstellt haben, sodass hier nur wenige Probleme zu erwarten sind.

Als wichtigste Quelle hat sich *DefinitelyTyped* etabliert. Hierbei handelt es sich um ein Repository, über das Sie Typdefinitionen für zahlreiche Bibliotheken beziehen können. Die Typdefinitionen werden in einem GitHub-Repository gepflegt und können selektiv über einen Paketmanager wie *Yarn* oder *NPM* installiert werden. Achten Sie bei der Installation einer Typdefinition darauf, dass Sie diese als `devDependency` installieren. Typdefinitionen werden nur während der Entwicklung und nicht zum Betrieb Ihrer Applikation verwendet.

Um die Typdefinition für React zu installieren, was *Create React App* automatisch für Sie übernimmt, führen Sie das Kommando `npm install -D @types/react` aus. Das Präfix `@types` steht hierbei für das DefinitelyTyped-Repository. Nach der Installation müssen Sie nichts weiter unternehmen, sondern können direkt mit den Typdefinitionen arbeiten, da TypeScript automatisch nach einem *@types*-Verzeichnis in den *node_modules* sucht, falls für eine bestimmte Bibliothek keine direkten Typdefinitionen vorhanden sind.

Mit diesem Wissen um TypeScript können Sie nun dazu übergehen, TypeScript Schritt für Schritt in Ihre Applikation zu integrieren.

4.4 TypeScript und React

Um TypeScript in Ihrer mit Create React App initialisierten Applikation zu verwenden, haben Sie zwei Möglichkeiten: Entweder Sie starten direkt mit TypeScript und geben die `--template typescript`-Option bei der Initialisierung Ihrer Applikation an, oder Sie fügen TypeScript zu Ihrem bestehenden Projekt hinzu. In den folgenden Abschnitten demonstriere ich Ihnen die Nutzung von TypeScript zunächst an einem unabhängigen Beispiel. Anschließend wird der jeweilige Aspekt der Beispielapplikation auf TypeScript umgestellt.

4.4.1 Basisfeatures

Die Grundlagen von TypeScript sind überall gleich, egal ob Sie mit React, Angular oder serverseitig mit Node.js arbeiten. Deshalb werfen wir zunächst einen Blick auf die Grundlagen von TypeScript, bevor Sie die React-spezifischen Teile kennenlernen.

Variablen

In TypeScript deklarieren Sie eine Variable wie in JavaScript, mit dem Unterschied, dass Sie noch einen optionalen Typ angeben können. Initialisieren Sie die Variable mit einem Wert, können Sie die Typzuweisung weglassen, da TypeScript automatisch den Typ des zugewiesenen Wertes übernimmt (*Type Inference*).

Für die Deklaration stehen Ihnen die Schlüsselworte `var`, `let` und `const` zur Verfügung, wobei Sie möglichst nur `let` und `const` verwenden sollten, da `var` nur Gültigkeitsbereiche auf Funktions- und nicht auf Blockebene unterstützt. Versuchen Sie so oft wie möglich, `const` zu verwenden, um eine versehentliche Neuzuweisung zu vermeiden. Erst wenn Sie einen Wert neu zuweisen müssen oder dies von vornherein wissen, verwenden Sie das `let`-Schlüsselwort. Die Typzuweisung bei einer `const`-Deklarationen ist vor allem bei Objekten und Arrays sinnvoll, in allen anderen Fällen können Sie sich auf die Type Inference von TypeScript verlassen.

`const` und `let` erzeugen Konstanten beziehungsweise Variablen im Block-Scope, sodass Sie sehr gute Kontrolle über die Gültigkeit haben. In Listing 4.5 sehen Sie ein Beispiel für eine Deklaration und gleichzeitige Initialisierung einer Variablen, bei der Sie zusätzlich den Typ angeben:

```
let title: string = 'Design Patterns';
```

Listing 4.5: Deklaration und Initialisierung von Variablen

Die Angabe des Typs schreiben Sie immer durch einen Doppelpunkt getrennt hinter den Namen der Variablen und vor das Gleichheitszeichen. Tabelle 4.1 enthält eine Übersicht der Datentypen von TypeScript.

Typ	Beschreibung
`boolean`	die Wahrheitswerte `true` und `false`
`number`	Zahlenwerte wie Ganzzahlen, Fließkommazahlen, aber auch `BigInt`
`string`	Zeichenketten
`array`	Entspricht dem Array-Typ von JavaScript.
`tuple`	ein Array mit fester Länge und fest zugeordneten Typen
`enum`	ein Enumerationstyp, bei dem Schlüssel auf Zahlen oder optional andere Werte gemappt werden
`unknown`	Der Inhalt einer Variablen dieses Typs kann von einem beliebigen Typ sein.
`any`	Schaltet die Typüberprüfung für eine Variable ab.
`void`	Kennzeichnet die Abwesenheit eines Werts, beispielsweise bei einer Funktion ohne Rückgabewert.
`null`	der `null`-Wert von JavaScript
`undefined`	der `undefined`-Wert von JavaScript

Tabelle 4.1: Basisdatentypen in TypeScript

Typ	Beschreibung
`never`	ein Wert, der nie auftritt. Ein Beispiel ist eine Funktion, die in jedem Fall eine Exception auslöst. Übergeben Sie bei der State-Initialisierung ein leeres Array, ist dieses vom Typ `never[]`.
`object`	Steht für ein JavaScript-Objekt.

Tabelle 4.1: Basisdatentypen in TypeScript (Forts.)

Funktionen

Funktionen nehmen bei der Entwicklung von React-Applikationen eine besondere Bedeutung ein. In einer React-Applikation haben Sie normalerweise deutlich häufiger mit Funktionen als mit Klassenkonstrukten zu tun. Funktionen können entweder als Arrow-Funktion, als benannte oder als anonyme Funktion auftreten. Für TypeScript ist die Signatur der Funktion relevant, also die Parameterliste und der Rückgabewert.

Listing 4.6 zeigt drei Beispiele für Funktionen:

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

const getFullname = function (firstname: string, lastname: string):
string {
  return `${firstname} ${lastname}`;
};

const greet = (name: string): string => {
  return `Hello ${name}`;
};
```

Listing 4.6: Beispiele für Funktionen in TypeScript

In Listing 4.6 sehen Sie Beispiele für

- eine *benannte Funktion*, also eine Funktion, die einen eigenen Namen hat,
- eine *anonyme Funktion*, also eine Funktion, die keinen Namen hat und die Sie einer Konstanten zuweisen, und schließlich für
- eine *Arrow-Funktion*.

Egal für welchen Typ von Funktion Sie sich entscheiden: Versuchen Sie, möglichst explizit zu sein, und geben Sie immer die Typen für die Parameter und den Rückgabewert an. Gerade der Rückgabewert kann Sie vor Flüchtigkeitsfehlern bewahren, wenn Sie beispielsweise ein `return`-Statement vergessen.

Klassen

Es gibt immer wieder Stellen, an denen es sich für Sie lohnt, auf Klassen zu setzen. Gerade für die Datenkapselung, die Implementierung bestimmter Businesslogik und nicht zuletzt für Klassenkomponenten kommen Klassen zum Einsatz.

Eine TypeScript-Klasse verhält sich sehr ähnlich wie eine Klasse in modernem JavaScript. So definieren Sie sie mit dem `class`-Schlüsselwort, gefolgt vom Namen der Klasse, der laut Namenskonvention mit einem Großbuchstaben beginnen sollte. Nach dem Klassennamen können Sie das Schlüsselwort `extends` und einen Klassennamen angeben, um von dieser Klasse zu erben, oder Sie nutzen das Schlüsselwort `implements` und geben ein Interface an, das die Klasse dann implementieren muss. Die Verwendung von Interfaces ist der erste große Unterschied zu herkömmlichem JavaScript.

In einer Klasse können Sie einen *Konstruktur* definieren. Das ist eine spezielle Methode, die aufgerufen wird, wenn Sie eine neue Instanz der Klasse mit `new` erzeugen. Der Name des Konstruktors lautet `constructor`. Im Konstruktor können Sie eine Parameterliste definieren. Diese Werte übergeben Sie bei der Instanziierung.

Neben dem Konstruktor können Sie in einer TypeScript-Klasse *Eigenschaften* und *Methoden* definieren. Methoden folgen bei der Angabe der Signatur den gleichen Regeln wie schon die Funktionen. Bei Eigenschaften und auch Methoden können Sie die Sichtbarkeit über die Zugriffsmodifikatoren `private`, `public` und `protected` festlegen.

Zugriffsmodifikatoren in TypeScript

Im Gegensatz zu nativem JavaScript unterstützt TypeScript Zugriffsmodifikatoren, mit denen Sie die Sichtbarkeit von Eigenschaften und Methoden einer Klasse beeinflussen können. TypeScript verfügt über die drei auch aus anderen Programmiersprachen bekannten Modifikatoren `private`, `protected` und `public`:

- **`private`:** Auf Eigenschaften und Methoden, die mit dem `private`-Modifikator ausgezeichnet sind, kann nur innerhalb der Klasse zugegriffen werden. Das bedeutet, dass sie nicht außerhalb, aber auch nicht in abgeleiteten Klassen verfügbar sind.
- **`protected`:** Eine Eigenschaft, die als `protected` markiert ist, kann in der Klasse und deren Subklassen verwendet werden.
- **`public`:** `public` ist der Standardmodifikator in TypeScript. Geben Sie keinen Modifikator an, ist die Eigenschaft oder Methode automatisch `public` und kann überall in Ihrer Applikation verwendet werden.

Ein weiterer Modifikator, den Sie im Zuge einer Klassendefinition verwenden können, ist `readonly`. Eigenschaften, die mit `readonly` ausgezeichnet sind, müssen im Konstruktor oder bei ihrer Deklaration initialisiert werden und können später nicht mehr verändert werden.

Definieren Sie im Konstruktor eine Parameterliste und möchten Sie diese Werte bestimmten Eigenschaften der Klasse zuweisen, führen Sie diese Operation direkt im Konstruktor durch. TypeScript sieht für diesen sehr häufig verwendeten Use-Case eine Abkürzung vor: Wenn Sie bei einem Parameter eine Kombination aus Zugriffsmodifikator, Eigenschaftsnamen und Typ angeben, weist TypeScript diesen Wert automatisch der angegebenen Klasseneigenschaft zu, und Sie müssen sich um nichts weiter kümmern. Diese Kurzschreibweise trägt den Namen *Parameter Properties*. Im folgenden Beispiel nutzen Sie die Parameter Properties für die Eigenschaften `firstname` und `lastname`.

In Listing 4.7 sehen Sie ein Beispiel für eine TypeScript-Klasse inklusive Instanziierung und Methodenaufruf:

```
class User {
  constructor(private firstname: string, private lastname: string) {}

  get fullname(): string {
    return `${this.firstname} ${this.lastname}`;
  }

  greet(greeting: string): string {
    return `${greeting} ${this.fullname}`;
  }
}

const klaus = new User('Klaus', 'Müller');
const greeting = klaus.greet('Hello');
console.log(greeting); // Hello Klaus Müller
```

Listing 4.7: Klasse in TypeScript

Generics

Generics machen es möglich, Funktionen, Klassen oder Typen so zu implementieren, dass sie mit unterschiedlichen Datentypen arbeiten können. Generics verfügen dabei über einen oder mehrere Platzhalter, die bei der Verwendung des Generics festgelegt werden. Damit können Sie wiederverwendbare und flexible Strukturen erstellen, die trotzdem typsicher sind.

Generics sind besonders hilfreich, wenn der Eingabetyp und der Ausgabetyp zusammenhängen – wie bei Funktionen, die etwas zurückgeben, das vom Eingabetyp abhängt.

In Listing 4.8 sehen Sie ein Beispiel für eine generische Klasse:

```
class Collection<T> {
  private items: T[] = [];

  add(item: T): void {
```

```
    this.items.push(item);
  }

  remove(item: T): void {
    this.items = this.items.filter(i => i !== item);
  }

  getAll(): T[] {
    return [...this.items];
  }

  first(): T | undefined {
    return this.items[0];
  }
}

const numberCollection = new Collection<number>();
numberCollection.add(1);
numberCollection.add(2);

const userCollection = new Collection<{ id: number; name: string }>();
userCollection.add({ id: 1, name: "Alice" });
userCollection.add({ id: 2, name: "Bob" });
```

Listing 4.8: Beispiel für eine generische Klasse

Im Beispiel erzeugen Sie zunächst eine Instanz der generischen `Collection`-Klasse mit dem Typ `number`. Dabei werden alle Typangaben mit dem Platzhalter `T` durch den Typ `number` ersetzt. Bei der zweiten Instanz belegen Sie den Platzhalter mit einem Objekttyp mit den Eigenschaften `id` und `name`.

Generics kommen in den Typdefinitionen von React häufiger vor. Ein populäres Beispiel ist `useState()`, eine generische Funktion, bei der Sie den Typ des States festlegen können.

Type Aliases vs. Interfaces

In TypeScript haben Sie mehrere Möglichkeiten, um Ihre eigenen Typen zu definieren. Mit Klassen haben Sie bereits eine dieser Möglichkeiten kennengelernt. Zwei weitere Varianten sind *Type Aliases* und *Interfaces*.

Interfaces in TypeScript

Interfaces beschreiben die Form eines Objekts und legen fest, welche Eigenschaften und Methoden vorhanden sein müssen. Sie dienen als Vertrag zwischen Codebausteinen und helfen dabei, klare und typsichere Strukturen zu definieren. Interfaces

beschreiben lediglich die Strukturen, nicht jedoch die Implementierung. Implementiert eine Klasse ein Interface (z. B. `class User implements Account`), muss die Klasse alle Anforderungen des Interface erfüllen.

Sie können zwar sowohl Type Aliases als auch Interfaces zur Angabe von Typen verwenden, also beispielsweise bei Variablendeklarationen oder in Funktionssignaturen, aber dennoch unterscheiden sich beide in einigen Punkten:

- Interfaces können Sie durch *Interface Merging* erweitern. Das heißt, definieren Sie ein Interface mehrmals, fügt TypeScript diese Definitionen zu einem Interface zusammen.
- Interfaces können mit dem `extends`-Schlüsselwort von anderen Interfaces erben. Bei Typen können Sie den `&`-Operator verwenden, um einen Typ als Basis zu verwenden und weitere Informationen hinzuzufügen und so einen neuen Typ zu erzeugen.
- Eine Klasse kann mit dem `implements`-Schlüsselwort ein Interface implementieren und muss in diesem Fall alle Eigenschaften und Methoden des Interfaces umsetzen.
- Type Aliases können nach ihrer Definition nicht mehr verändert werden.

Für einfache Fälle und wenn Sie nicht sicherstellen müssen, dass eine Klasse ein Interface implementieren muss, reicht in den meisten Fällen ein Type Alias aus. Mit diesem Grundwissen in TypeScript wenden wir uns jetzt etwas React-spezifischeren Themen zu.

4.4.2 Funktionskomponenten

Funktionskomponenten in TypeScript

- Bei Funktionskomponenten sind vor allem die Typen der Props und des Rückgabewerts relevant.
- Der Rückgabetyp ist `ReactNode`.
- Für Funktionskomponenten können Sie den generischen `React.FC`-Typ nutzen und ihm bei Bedarf die Props-Struktur übergeben:

```
import React from 'react';

type Props = {
  title: string;
};

const Headline: React.FC<Props> = ({ title }) => {
  return <h1>{title}</h1>
}

export default Headline;
```

Listing 4.9: Typisierung bei einer Funktionskomponente

Bei den Funktionskomponenten wirkt sich TypeScript hauptsächlich in der Signatur der Funktion aus. React, beziehungsweise die Typdefinition von React, sieht den generischen Typ `React.FC` für Funktionskomponenten vor. Dieser Typ ist einige Zeit lang in die Kritik geraten, da er immer Kindkomponenten für eine Komponente vorsieht, obwohl die Komponente diese nicht verwendet, was teilweise etwas irreführend war. Dieses Problem ist jedoch mittlerweile behoben worden, und so steht dem Einsatz dieses Typs nichts im Wege.

Bei einer Funktionskomponente sollten Sie einen Typ für die Props definieren und diesen als separaten Type Alias innerhalb der Datei ablegen. Sofern Sie die Props nicht an anderer Stelle in Ihrer Applikation nutzen, sollten Sie diese auch nicht unnötig exportieren. Als Typ für die Funktionskomponente selbst nutzen Sie `React.FC`, der als generischer Typ implementiert ist und die Struktur der Props akzeptiert. `React.FC` ist dafür verantwortlich, den korrekten Rückgabetyp für die Funktionskomponente zu definieren. Als Beispiel für eine typische Funktionskomponente sehen Sie in Listing 4.10 die `BooksListItem`-Komponente. Diese ist für die Anzeige eines Datensatzes in einer Liste zuständig. Diese Komponente erhält den Datensatz als Prop und stellt den Titel des Datensatzes in einem `li`-Element dar:

```
import React from 'react';
import type { Book } from './Book';

type Props = {
  book: Book;
};

const BooksListItem: React.FC<Props> = ({ book }) => {
  return <li>{book.title}</li>;
};

export default BooksListItem;
```

Listing 4.10: Typisierte Funktionskomponente (»src/BooksListItem.tsx«)

In dieser Komponente verweisen Sie in den Props auf den Typ `Book`. Solche Typen benötigen Sie in einer Applikation in der Regel mehr als nur einmal, also sollten Sie sie in einer separaten Datei speichern, in diesem Fall im *src*-Verzeichnis mit dem Namen *Book.ts*. Den Quellcode dieser Datei sehen Sie in Listing 4.11:

```
export type Book = {
  id: number;
  title: string;
  author: string;
  isbn: string;
  rating: number;
};
```

Listing 4.11: Book-Type (»src/Book.ts«)

Die `BooksItemList`-Komponente muss von ihrer Elternkomponente den Datensatz erhalten, den sie anzeigen soll. Die Elternkomponente wiederum muss sich dann auch um den State der Liste kümmern.

Der State-Hook

Schnellstart

Die `useState`-Komponente ist als generische Komponente definiert. Sie können den Typ des States also gesondert angeben. Alternativ können Sie sich auch auf die *Type Inference* von TypeScript verlassen, also auf die Fähigkeit von TypeScript, einen Typ aus gegebenen Typangaben abzuleiten. In diesem Fall leitet TypeScript den Typ vom initialen Wert des States ab:

```
const [state, setState] = useState<string[]>([]);
```

Listing 4.12: Die Syntax der »useState«-Funktion in TypeScript

Der *State-Hook* ist der erste Basis-Hook, bei dem TypeScript relevant wird, da Sie beim *Effect-Hook* keine Typen angeben müssen. Die Typdefinition der `useState()`-Funktion sieht vor, dass es sich dabei um eine generische Funktion handelt. Das bedeutet, dass Sie in den spitzen Klammern nach dem Funktionsnamen den Typ angeben können, mit dem die Komponente arbeitet. Die `BooksList`-Komponente aus Listing 4.13 verwaltet den State und kümmert sich um das Rendering der Kindkomponenten:

```
import React, { useState } from 'react';
import BooksListItem from './BooksListItem';
import type { Book } from './Book';

const initialBooks: Book[] = [
  {
    id: 1,
    title: 'JavaScript - Das umfassende Handbuch',
    author: 'Philip Ackermann',
    isbn: '978-3836286299',
    rating: 5,
  },…
];

const BooksList: React.FC = () => {
  const [books, setBooks] = useState<Book[]>(initialBooks);

  return (
    <ul>
      {books.map((book) => (
```

```
        <BooksListItem key={book.id} book={book} />
      ))}
    </ul>
  );
};

export default BooksList;
```

Listing 4.13: Verwaltung des lokalen States einer Komponente mit TypeScript (»src/BooksList.tsx«)

Beim Aufruf der `useState()`-Funktion ist die Angabe des Typs in diesem Fall optional, da TypeScript den verwendeten Typ aus dem Initialwert ableiten kann. Ist dies nicht der Fall, können Sie in TypeScript ein Array aus `Book`-Objekten entweder über die gebräuchlichere Variante als `Book[]` definieren oder Sie verwenden die generische Array-Schreibweise `Array<Book>`.

Im letzten Schritt müssen Sie die `BooksList`-Komponente noch in Ihre `App`-Komponente einbinden, damit React die Liste korrekt rendert. Den Code dieser Komponente finden Sie in Listing 4.14:

```
import type React from 'react';
import './App.css';
import BooksList from './BooksList';

const App: React.FC = () => {
  return (
    <div>
      <h1>Bücherverwaltung</h1>
      <BooksList />
    </div>
  );
};

export default App;
```

Listing 4.14: Integration der »BooksList«-Komponente in die »App«-Komponente (»src/App.tsx«)

4.5 Zusammenfassung

Dieses Kapitel hat Ihnen gezeigt, dass ein Typsystem wie TypeScript eine gute Ergänzung für React und sein gesamtes Ökosystem ist:

- Durch die Verwendung eines Typsystems können Sie die Qualität und die Lesbarkeit Ihres Quellcodes verbessern. Das gilt vor allem bei umfangreicheren Applikationen.

- Eine Alternative zum weitverbreiteten *TypeScript* ist das von Facebook entwickelte *Flow*.
- Ein Typsystem bietet Ihnen einen Basissatz an Typen und ermöglicht es Ihnen, eigene Typen in Form von Klassen, Interfaces oder *Type Aliases* zu definieren.
- Bei der Variablendeklaration und in der Signatur von Funktionen können Sie Typen angeben, sodass der TypeScript-Compiler die Einhaltung der Schnittstelle sicherstellen kann.
- In vielen Fällen können Sie auf die explizite Angabe von Typen verzichten, da TypeScript mit seinem Type-Inference-Feature versucht, den passenden Typ zu ermitteln. Versuchen Sie jedoch stets, so explizit wie möglich zu sein, auch wenn das bedeutet, dass Sie etwas mehr Quellcode schreiben müssen. TypeScript hilft Ihnen in diesem Fall, Flüchtigkeitsfehler zu vermeiden.
- *DefinitelyTyped* stellt Ihnen für die meisten Bibliotheken von Drittanbietern Typdefinitionen zur Verfügung, die als NPM-Pakete installiert werden können.
- Funktionskomponenten versehen Sie mit einer typisierten Parameterliste sowie mit einem Rückgabetyp.
- React liefert Typdefinitionen für die Hook-API, sodass Sie beispielsweise für die generische Funktion `useState()` den passenden Typ angeben können.

Im nächsten Kapitel steigen Sie tiefer in die Möglichkeiten von React-Komponenten ein und lernen den Lebenszyklus genauer kennen. Sie erfahren, wie Sie Daten vom Server beziehen können, und sehen, wie Sie verschiedene Architekturmuster umsetzen können. Außerdem erfahren Sie, was es mit der Context-API von React auf sich hat.

Kapitel 5

Ein Blick hinter die Kulissen – weiterführende Themen

Sie wissen bereits, wie Sie Komponenten implementieren, States verwalten und eine Komponentenhierarchie aufbauen. In diesem Kapitel erfahren Sie mehr über den Lebenszyklus einer Komponente, die Architektur von Komponenten und wie Sie auf Daten unabhängig vom Komponentenbaum zugreifen können.

Eine Komponentenhierarchie sorgt für eine klare Struktur. Der Datenfluss von den Eltern- zu den Kindkomponenten erfolgt dabei über Props. Zusätzlich können Elternkomponenten Funktionen per Props an ihre Kindkomponenten übergeben, um die Kommunikation in die andere Richtung zu ermöglichen. Mit diesen Mechanismen lassen sich bereits viele Anwendungsfälle abdecken. In React haben sich jedoch noch zahlreiche weitere Patterns etabliert, mit deren Hilfe Sie Ihre React-Applikation so strukturieren können, dass sie auch bei größerem Funktionsumfang noch übersichtlich bleibt.

Bevor wir uns jedoch den eigentlichen Patterns widmen, lernen Sie zunächst den Lebenszyklus einer Komponente kennen.

5.1 Der Lebenszyklus einer Komponente

Eine Komponente durchläuft in einer Applikation einen Lebenszyklus, den Sie in die folgenden drei Stufen unterteilen können:

- **Mount:** Der erste Schritt im Leben einer Komponente besteht darin, dass sie in den Komponentenbaum eingehängt (»gemountet«) und damit gerendert wird. Während dieser Stufe des Lebenszyklus führen Sie meist Initialisierungsaufgaben durch, beispielsweise das Laden von Daten vom Server.
- **Update:** Die wenigsten Komponenten sind rein statisch. Meist verwalten sie ihren eigenen State oder erhalten dynamische Daten über Props. Ändern sich diese Informationen, sorgt das für ein Rerendern der Komponente. Auf solche Aktualisierungen können Sie gezielt reagieren.
- **Unmount:** Der letzte Abschnitt im Lebenszyklus einer Komponente ist das Aushängen der Komponente aus dem Komponentenbaum. Sie haben die Möglichkeit, auch an dieser Stelle Logik auszuführen. Meist nutzen Sie diese Gelegenheit, um Ressourcen wieder freizugeben, beispielsweise eine geöffnete WebSocket-Verbindung.

5.2 Der Lebenszyklus einer Funktionskomponente mit dem Effect-Hook

Der Lebenszyklus einer Komponente

- Mit `useEffect()` können Sie in alle Phasen des Lebenszyklus einer Komponente eingreifen.
- Es kann mehrere `useEffect()`-Aufrufe pro Komponente geben.
- Die Syntax von `useEffect()` sieht so aus:

```
useEffect(effectFunction, dependencies)
```

Der Lebenszyklus sieht wie folgt aus:

- **Mount:** Sie übergeben eine Callback-Funktion, die beim Mounten ausgeführt werden soll. Außerdem übergeben Sie als zweites Argument ein leeres Array.
- **Update:** Neben der Callback-Funktion übergeben Sie entweder kein zweites Argument, dann wird der Callback bei jeder Aktualisierung ausgeführt; oder Sie geben ein Array an: Dann wird die Funktion nur ausgeführt, wenn sich eine der angegebenen Abhängigkeiten geändert hat.
- **Unmount:** Auf das Aushängen Ihrer Komponente aus dem Komponentenbaum können Sie reagieren, indem Sie aus der Callback-Funktion wiederum eine Funktion zurückgeben. Diese wird ausgeführt, wenn die Komponente entfernt wird.

Bisher kennen Sie nur die Komponentenfunktion selbst, aber Sie haben noch nicht in den Lebenszyklus der Komponente eingegriffen, zumindest nicht bewusst. React führt als ersten Schritt im Lebenszyklus die Komponentenfunktion selbst aus. Hier sollten Sie jedoch auf jeglichen Seiteneffekt verzichten, da React in der Lage ist, den Renderprozess zu unterbrechen, später fortzusetzen oder ihn vollständig abzubrechen.

Beispiele für einen *Seiteneffekt* sind Operationen, die nicht direkt mit dem Rendern der Komponente zu tun haben, wie beispielsweise die Kommunikation mit einem Webserver, um Daten zu laden, oder das Setzen eines Timeouts oder Intervalls. Würden Sie einen solchen Seiteneffekt direkt in die Komponentenfunktion platzieren und würde der Rendervorgang abgebrochen und erneut gestartet, dann würde der Seiteneffekt ein zweites Mal ausgeführt, was im besten Fall keinen weiteren Schaden anrichtet, aber auch zu schwerwiegenden Problemen führen kann.

5.2.1 Mount – das Einhängen einer Komponente

Nachdem Sie jetzt wissen, dass Sie keine Seiteneffekte direkt in der Komponentenfunktion auslösen dürfen, benötigen Sie eine Stelle, an der Sie dies dürfen. Als Beispiel nutzen wir hier wieder die Bücherliste aus dem vorherigen Beispiel, allerdings in einer deutlich einfacheren Variante. Die Komponente verwaltet ihren eigenen State und legt dort die

Datensätze als Array ab. Initial starten Sie mit einem leeren Array. Nachdem die Komponente geladen ist, füllen Sie den State mit Daten. In Listing 5.1 sehen Sie den Code der Komponente:

```
import { useState, useEffect } from 'react';
import type { Book } from './Book';

const booksData: Book[] = [
  {
    id: 1,
    title: 'JavaScript - das umfassende Handbuch',
    author: 'Philip Ackermann',
    isbn: '978-3836286299',
    rating: 5,
  },
];

const BooksList: React.FC = () => {
  const [books, setBooks] = useState<Book[]>([]);

  useEffect(() => {
    setTimeout(() => {
      setBooks(booksData);
    }, 2000);
  }, []);

  if (books.length === 0) {
    return <div>Keine Bücher gefunden</div>;
  } else {
    return (
      <table>
        <thead>
          <tr>
            <th>Titel</th>
            <th>Autor</th>
            <th>ISBN</th>
          </tr>
        </thead>
        <tbody>
          {books.map((book) => (
            <tr key={book.id}>
              <td>{book.title}</td>
              <td>{book.author}</td>
              <td>{book.isbn}</td>
```

```
            </tr>
          ))}
        </tbody>
      </table>
    );
  }
}

export default BooksList;
```

Listing 5.1: Asynchrone Befüllung des Komponentenstates während des Mountens (»src/BooksList.tsx«)

Die `BooksList`-Komponente rendert im ersten Schritt das leere Array aus dem initialen State. Das bedeutet, dass Sie zunächst im Browser eine Ansicht wie in Abbildung 5.1 erhalten.

Abbildung 5.1: Initiale Darstellung der »BooksList«-Komponente

Der neue Aspekt in der Komponente ist der Aufruf der `useEffect()`-Funktion. Beachten Sie an dieser Stelle, dass die Funktion mit zwei Argumenten aufgerufen wird: Das erste ist eine Callback-Funktion und das zweite ein leeres Array. Das leere Array spielt hier eine bedeutende Rolle. Es legt fest, dass die Callback-Funktion, die Sie als erstes Argument übergeben, nur einmalig beim Mounten der Komponente ausgeführt wird. Lassen Sie dieses Array weg, führt React die Callback-Funktion bei jedem Update aus. Was das für Konsequenzen hat, erfahren Sie im nächsten Abschnitt.

Innerhalb der Callback-Funktion, die Sie an `useEffect()` übergeben, setzen Sie zunächst einen Timeout über zwei Sekunden. Dieser dient lediglich dazu, dass Sie die Auswirkung des Mount-Hooks besser sehen. In der Timeout-Funktion überschreiben Sie den aktuellen State der Komponente mit einem neuen Array mit einem Datensatz, den Ihre Komponente anschließend anzeigt. Im Browser sehen Sie nach diesen zwei Sekunden dann die finale Darstellung wie in Abbildung 5.2.

Abbildung 5.2: Aktualisierte Darstellung nach dem Mount-Hook

Normalerweise ist die Zeitspanne zwischen diesen beiden Ansichten sehr kurz. Aber dennoch sollten Sie sich bewusst sein, dass Ihre Benutzer beide Varianten sehen. Diese Tatsache können Sie nutzen und beispielsweise einen Loading-Indicator oder Ähnliches anzeigen, um Ihre Benutzer darüber zu informieren, dass die Daten bald angezeigt werden. Im besten Fall nehmen die Benutzer die Ladeanzeige nicht wahr. Sollte die Operation doch einmal etwas länger dauern, sind die Benutzer dann aber informiert, dass im Hintergrund noch eine Aktion ausgeführt wird.

Implementierung eines Loading-Indicators

Wollen Sie einen Loading-Indicator umsetzen, müssen Sie mit einer Kombination aus `useEffect()` und `useState()` arbeiten. Im State halten Sie fest, ob die asynchrone Operation aktiv ist und deshalb der Loading-Indicator angezeigt werden soll. Der Standardwert dieses States ist `true`. Die Komponente befindet sich also initial schon im Ladezustand und rendert eine entsprechende Information. Im `useEffect()`-Aufruf findet dann die eigentliche Operation statt. Ist diese abgeschlossen, setzen Sie den Loading-Indicator wieder auf den Wert `false`. Wie das konkret im Code aussehen kann, sehen Sie in Listing 5.2:

```
import { useState, useEffect } from 'react';
import type { Book } from './Book';

const booksData: Book[] = [...];

const BooksList: React.FC = () => {
  const [books, setBooks] = useState<Book[]>([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    setTimeout(() => {
      setBooks(booksData);
```

```
      setLoading(false);
    }, 2000);
  }, []);

  if (loading) {
    return <div>Laden...</div>;
  }

  if (books.length === 0) {
    return <div>Keine Bücher gefunden</div>;
  } else {
    return (
      <table>...</table>
    );
  }
};

export default BooksList;
```

Listing 5.2: »BooksList«-Komponente mit Loading-State

Ein weiterer Aspekt des Effekt-Hooks beim Mounten der Komponente ist, dass die Komponentenfunktion in diesem Fall zweimal ausgeführt wird. Beim ersten Mal wird die Komponente mit der Meldung gerendert, dass keine Daten vorhanden sind, und das zweite Mal zeigt React dann den Datensatz an. Normalerweise sind solche Komponentenfunktionen leichtgewichtig, sodass dies nicht ins Gewicht fällt, vor allem dann nicht, wenn Sie auf Seiteneffekte verzichten. Sie sollten sich dennoch bewusst sein, dass die Funktion öfter als nur einmal aufgerufen wird.

5.2.2 Update – das Aktualisieren der Komponente

Sobald sich der State oder die Props ändern, zeichnet React die Komponente neu. In diesem Fall tritt die Komponente in die Update-Stufe ihres Lebenszyklus. Mit dem Effect-Hook haben Sie die Möglichkeit, hierauf zu reagieren. Relevant wird diese Möglichkeit beispielsweise, wenn Sie auf Aktualisierungen des States reagieren müssen. State-Updates sind in React asynchron. Sie können also keine Logik im Sinne von »Führe diese Funktion aus, nachdem der State aktualisiert wurde« direkt daran knüpfen. Und an genau dieser Stelle kommt der Effekt-Hook ins Spiel. Der Code aus Listing 5.3 verdeutlicht diesen Zusammenhang:

```
import { useState, useEffect } from 'react';
import type { Book } from './Book';
```

```
const booksData: Book[] = […];

const BooksList: React.FC = () => {
  const [books, setBooks] = useState<Book[]>([]);

  useEffect(() => {
    setTimeout(() => {
      setBooks(booksData);
    }, 2000);
  }, []);

  useEffect(() => {
    console.log('Elemente im State: ', books.length);
    console.log(
      'Tabellenzeilen: ',
      document.querySelectorAll('tbody tr').length
    );
  });

  if (books.length === 0) {
    return <div>Keine Bücher gefunden</div>;
  } else {
    return (
      <table>
        <thead>
          <tr>
            <th>Titel</th>
            <th>Autor</th>
            <th>ISBN</th>
          </tr>
        </thead>
        <tbody>
          {books.map((book) => (
            <tr key={book.id}>
              <td>{book.title}</td>
              <td>{book.author}</td>
              <td>{book.isbn}</td>
            </tr>
          ))}
        </tbody>
      </table>
    );
  }
```

Inhalt

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

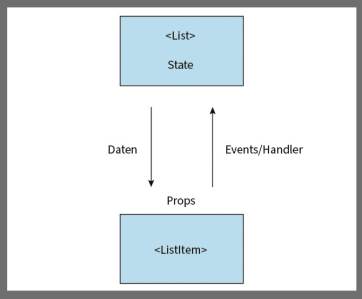
Z

React

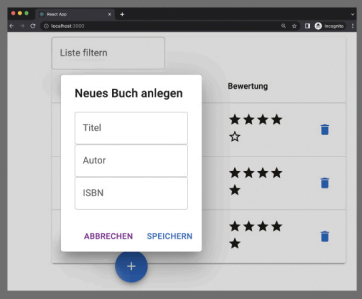
»Nutzen Sie das Potenzial von React!«

So geht moderne Frontend-Entwicklung mit React

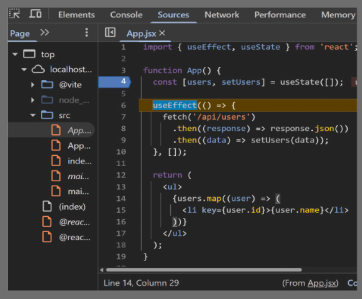
Skalierbar, flexibel, performant: React ist die führende JavaScript-Bibliothek für moderne Benutzeroberflächen. Dieses Buch begleitet Sie durch alle Phasen der Entwicklung – von den Grundlagen bis zur professionellen Anwendung. Anhand vieler Beispiele lernen Sie den modernen Standard-Stack rund um Next.js, React Native und Redux kennen.



Konzepte verstehen



An Beispielen lernen



Eigene Apps entwickeln

Die ersten Schritte mit React

Erfahren Sie, was React ausmacht und wie Sie moderne Benutzeroberflächen entwickeln. Lernen Sie die grundlegende Struktur einer Applikation und die Grundlagen von React kennen.

React verstehen und optimal einsetzen

Von Komponenten, JSX und Hooks über TypeScript, Formulare und Styling bis hin zu Serverkommunikation mit TanStack Query: Lernen Sie die Werkzeuge kennen, die in der Praxis wirklich zählen – inklusive Best Practices für Ihre Projekte.

Professionelle und skalierbare Anwendungen entwickeln

Ob Single-Page-Applikation, Unternehmensanwendung oder mobile App: Entwickeln Sie performante und skalierbare Anwendungen. Themen wie Testing, Performance, Routing, Authentifizierung und KI-Anwendungen runden Ihr Wissen ab.

Der Code aller Beispielprojekte steht zum Download bereit.

Sebastian Springer ist Web Developer bei Rohde & Schwarz. Als Dozent lehrt er JavaScript an der Technischen Hochschule Rosenheim. Er veröffentlicht regelmäßig Artikel in Fachzeitschriften und ist ein gefragter Speaker auf Konferenzen.

Aus dem Inhalt

- Erste Schritte mit React
- Typsicherheit mit TypeScript
- Styling von Komponenten
- Absicherung durch automatisierte Tests
- Interaktion über Formulare und Validierung
- Performance-Optimierung
- State-Management mit Redux
- Asynchronität und Serverkommunikation mit TanStack Query
- Routing und Navigation in React Applikationen
- Server Side Rendering und Fullstack React mit Next.js
- Serverkommunikation mit GraphQL und dem Apollo Client

Rheinwerk Computing

Gedruckt in Deutschland
Mineralölfreie Druckfarben
Zertifiziertes Papier

€ 49,90 [D] € 51,30 [A]

Für Windows, macOS und Linux
Programmierung
ISBN 978-3-367-11543-3