

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  
        argument to be a function that returns a number')  
    }  
  }  
}
```



Sebastian Springer

React

Das umfassende Handbuch

- ▶ Für Einsteiger und Fortgeschrittene
- ▶ Komponentenarchitektur, Hooks, State-Management und Formulgestaltung
- ▶ 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 benötigten 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;
```

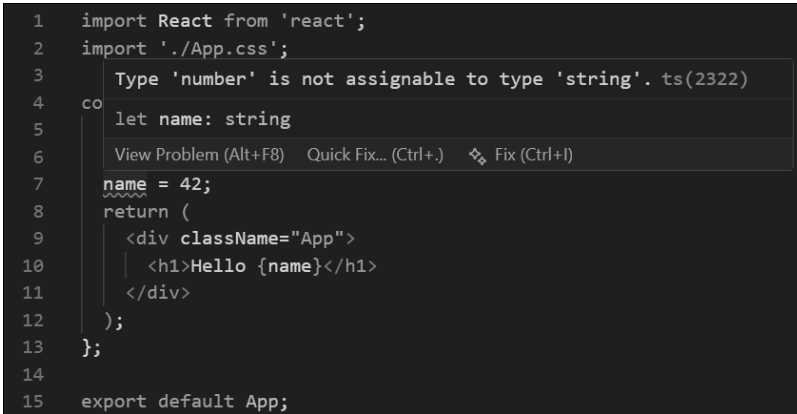
Listing 4.2: TypeScript-Quellcode (»src/App.tsx«)

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-Variablen 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.



```

1  import React from 'react';
2  import './App.css';
3
4  const App = () => {
5      let name: string;
6      name = 42;
7      return (
8          <div className="App">
9              <h1>Hello {name}</h1>
10             </div>
11         );
12     };
13
14     export default App;

```

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, so dass 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
<code>boolean</code>	die Wahrheitswerte <code>true</code> und <code>false</code>
<code>number</code>	Zahlenwerte wie Ganzzahlen, Fließkommazahlen, aber auch <code>BigInt</code>
<code>string</code>	Zeichenketten
<code>array</code>	Entspricht dem <code>Array</code> -Typ von JavaScript.
<code>tuple</code>	ein <code>Array</code> mit fester Länge und fest zugeordneten Typen
<code>enum</code>	ein Enumerationstyp, bei dem Schlüssel auf Zahlen oder optional andere Werte gemappt werden
<code>unknown</code>	Der Inhalt einer Variablen dieses Typs kann von einem beliebigen Typ sein.
<code>any</code>	Schaltet die Typüberprüfung für eine Variable ab.
<code>void</code>	Kennzeichnet die Abwesenheit eines Werts, beispielsweise bei einer Funktion ohne Rückgabewert.
<code>null</code>	der <code>null</code> -Wert von JavaScript
<code>undefined</code>	der <code>undefined</code> -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 <code>never[]</code> .
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 *Konstruktor* 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 Ausgabotyp 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 Rückgabotyp ist `ReactDOM`.
- 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 Rückgabetyt 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 Rückgabetyt.
- 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>
    );
  }
};
```

```
        </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.

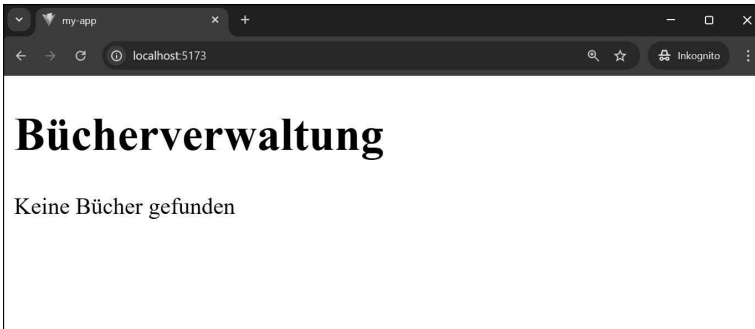


Abbildung 5.1: Initiale Darstellung der »BooksList«-Komponente

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.querySelector('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

Materialien zum Buch	19
Geleitwort des Fachgutachters	21
Vorwort	23
1 Die ersten Schritte mit React	27
1.1 Was ist React?	27
1.1.1 Single-Page-Applikationen	28
1.1.2 Die Geschichte von React	29
1.2 Warum React?	35
1.2.1 Der Release-Zyklus	36
1.3 Die wichtigsten Begriffe und Konzepte der React-Welt	37
1.3.1 Komponenten und Elemente	37
1.3.2 Der Datenfluss	40
1.3.3 Der Renderer	42
1.3.4 Der Reconciler	42
1.4 Ein Blick in das React-Universum	44
1.4.1 Das State-Management	45
1.4.2 Der Router	45
1.4.3 Material UI	45
1.4.4 Vitest	45
1.5 Thinking in React	46
1.5.1 Die Oberfläche in eine Komponentenhierarchie zerlegen	46
1.5.2 Eine statische Version in React implementieren	46
1.5.3 Den minimalen UI State bestimmen	47
1.5.4 Den Speicherort des States bestimmen	47
1.5.5 Den inversen Datenfluss modellieren	47
1.6 Codebeispiele	47
1.7 Zusammenfassung	48
2 Die ersten Schritte im Entwicklungsprozess	49
2.1 Schnellstart	49
2.1.1 Die Initialisierung	50
2.2 Playgrounds für React	51
2.2.1 CodePen – ein Playground für die Webentwicklung	51

- 2.3 **Lokale Entwicklung** 54
 - 2.3.1 React in eine HTML-Seite einbinden 54
- 2.4 **Der Einstieg in die Entwicklung mit React** 58
 - 2.4.1 Technische Voraussetzungen 59
 - 2.4.2 Das Build-Werkzeug Vite 60
 - 2.4.3 Alternativen zu Vite 67
 - 2.4.4 Vite Scripts 68
 - 2.4.5 Serverkommunikation im Entwicklungsbetrieb 71
- 2.5 **Die Struktur der Applikation** 73
- 2.6 **Fehlersuche in einer React-Applikation** 74
 - 2.6.1 Arbeiten mit den React Developer Tools 76
- 2.7 **Die Applikation bauen** 77
- 2.8 **Zusammenfassung** 78

- 3 Die Grundlagen von React** 79
 - 3.1 **Vorbereitung** 79
 - 3.2 **Einstieg in die Applikation** 80
 - 3.2.1 »main.jsx« – das Rendering der Applikation 80
 - 3.2.2 App.jsx – die Wurzelkomponente 83
 - 3.3 **Funktionskomponenten** 84
 - 3.3.1 Eine Komponente pro Datei 87
 - 3.4 **JSX – Strukturen in React definieren** 92
 - 3.4.1 Ausdrücke in JSX 95
 - 3.4.2 Iterationen – Schleifen in Komponenten 98
 - 3.4.3 Bedingungen in Komponenten 100
 - 3.5 **Props – Informationsfluss in einer Applikation** 103
 - 3.5.1 Props und Kindkomponenten 104
 - 3.5.2 Typsicherheit von Props zur Laufzeit 106
 - 3.6 **Lokaler State** 109
 - 3.7 **Event-Binding – Reaktion auf Benutzerinteraktionen** 111
 - 3.7.1 Auf Events reagieren 111
 - 3.7.2 Arbeiten mit Event-Objekten 117
 - 3.8 **Immutability** 121
 - 3.9 **Zusammenfassung** 125

4	Typsicherheit in React-Applikationen mit TypeScript	127
4.1	Was bringt ein Typsystem?	127
4.2	Die verschiedenen Typsysteme	128
4.3	TypeScript in einer React-Applikation einsetzen	129
4.3.1	TypeScript in eine React-Applikation einbinden	129
4.3.2	Konfiguration von TypeScript	132
4.3.3	Die wichtigsten Features von TypeScript	133
4.3.4	Typdefinitionen – Informationen über Drittanbieter-Software	133
4.4	TypeScript und React	134
4.4.1	Basisfeatures	134
4.4.2	Funktionskomponenten	140
4.5	Zusammenfassung	143
5	Ein Blick hinter die Kulissen – weiterführende Themen	145
5.1	Der Lebenszyklus einer Komponente	145
5.2	Der Lebenszyklus einer Funktionskomponente mit dem Effect-Hook	146
5.2.1	Mount – das Einhängen einer Komponente	146
5.2.2	Update – das Aktualisieren der Komponente	150
5.2.3	Unmount – das Aufräumen am Ende des Lebenszyklus	154
5.3	Serverkommunikation	158
5.3.1	Serverimplementierung	158
5.3.2	Serverkommunikation mit der Fetch-API	160
5.3.3	Umgebungsvariablen bei der Serverkommunikation verwenden	165
5.4	Container Components	167
5.4.1	Auslagern von Logik in eine Container Component	168
5.4.2	Einbindung der Container Component	170
5.4.3	Implementierung der Presentational Component	171
5.5	Higher-Order Components	172
5.5.1	Eine einfache Higher-Order Component	173
5.5.2	Einbindung einer Higher-Order Component in die BooksList-Komponente	176
5.5.3	Einbindung der Higher-Order Component	177

- 5.6 **Render Props** 178
 - 5.6.1 Alternative Namen für Render Props 180
 - 5.6.2 Integration der Render Props in die Applikation 182
- 5.7 **Kontext** 184
 - 5.7.1 Die Context-API 184
 - 5.7.2 Einsatz der Context-API in der Beispielapplikation 188
- 5.8 **Fragments** 192
- 5.9 **Zusammenfassung** 194

- 6 Serverkommunikation mit React** 195
 - 6.1 **Trennen von Komponente und Kommunikation** 197
 - 6.1.1 Umsetzung von API-Funktionen 197
 - 6.1.2 Lesender Serverzugriff 198
 - 6.1.3 Schreibende Serverzugriffe 199
 - 6.2 **Bibliotheken für die Serverkommunikation** 206
 - 6.3 **Validierung der Serverdaten mit Zod** 207
 - 6.4 **Daten mit TanStack Query vom Server laden** 209
 - 6.4.1 Setup von TanStack Query 210
 - 6.4.2 Daten vom Server laden 211
 - 6.4.3 TanStack Query und Suspense for Data Fetching 213
 - 6.4.4 Daten modifizieren 215
 - 6.4.5 Das Caching von TanStack Query 218
 - 6.5 **Zusammenfassung** 219

- 7 Formulare in React** 221
 - 7.1 **Uncontrolled Components** 221
 - 7.1.1 Der Umgang mit Referenzen in React 222
 - 7.2 **Controlled Components** 234
 - 7.2.1 Synthetic Events 240
 - 7.3 **Der Upload von Dateien** 240
 - 7.4 **Formularhandling mit React Hook Form** 247
 - 7.4.1 Formularvalidierung mit React Hook Form 251
 - 7.4.2 Formularvalidierung mit einem Schema 254
 - 7.4.3 Styling des Formulars 256
 - 7.5 **Zusammenfassung** 258

8	Die Hooks-API von React	259
8.1	Ein erster Überblick	260
8.1.1	Die drei Basis-Hooks	260
8.1.2	Weitere Bestandteile der Hooks-API	261
8.2	»useReducer« – der Reducer Hook	263
8.2.1	Die Reducer-Funktion	265
8.2.2	Actions und Dispatching	265
8.2.3	Asynchronität im Reducer-Hook	266
8.3	»useCallback« – Memoisieren von Funktionen	271
8.4	»useMemo« – Memoisieren von Objekten	273
8.5	»useRef« – Referenzen und immutable Values	275
8.5.1	Formularhandling mit dem Ref-Hook	275
8.5.2	Werte mit dem Ref-Hook zwischenspeichern	276
8.6	»useImperativeHandle« – Steuerung von ForwardRefs	277
8.6.1	ForwardRefs	278
8.6.2	Der ImperativeHandle-Hook	280
8.7	»useLayoutEffect« – die synchrone Alternative zu useEffect	281
8.8	»useDebugValue« – Debugging-Informationen in den React Developer Tools	282
8.9	»useDeferredValue« – Updates nach Priorität durchführen	283
8.10	»useTransition« – die Priorität von Operationen heruntersetzen	287
8.11	»useId« – eindeutiger Identifier erzeugen	289
8.12	»useEffectEvent« – stabile Callbacks für Effekte und Event-Logik	290
8.13	»useOptimistic« – optimistische Updates ohne komplexes State-Management	292
8.14	Bibliotheks-Hooks	295
8.14.1	»useSyncExternalStore«	295
8.14.2	»useInsertionEffect«	295
8.15	Custom Hooks	296
8.15.1	Ein Beispiel für einen Custom Hook	296
8.16	Rules of Hooks – was Sie beachten sollten	298
8.16.1	Regel #1: Hooks nur auf oberster Ebene ausführen	298
8.16.2	Regel #2: Hooks dürfen nur in Funktionskomponenten oder Custom Hooks verwendet werden	299
8.17	Zusammenfassung	300

9	Styling von React-Komponenten	301
9.1	CSS-Import	301
9.1.1	Die Vor- und Nachteile des CSS-Imports	303
9.1.2	Umgang mit Klassennamen	305
9.1.3	Verbesserte Behandlung von Klassennamen mit der »clsx«-Bibliothek	307
9.1.4	Verwendung von Sass als CSS-Präprozessor	309
9.2	Inline-Styling	311
9.3	CSS-Module	313
9.4	CSS in JavaScript mit Emotion	316
9.4.1	Emotion installieren	316
9.4.2	Arbeiten mit der »css«-Prop	317
9.4.3	Der styled-Ansatz von Emotion	319
9.4.4	Pseudoselektoren in Emotion Styled Components	321
9.4.5	Dynamisches Styling	322
9.4.6	Weitere Features von Emotion	324
9.5	Tailwind	325
9.5.1	Tailwind installieren und einbinden	325
9.6	Zusammenfassung	327
10	Eine React-Applikation durch Tests absichern	329
10.1	Die ersten Schritte mit Vitest	331
10.1.1	Installation und Ausführung	332
10.1.2	Organisation der Tests	334
10.1.3	Vitest – die Grundlagen	334
10.1.4	Aufbau eines Tests – Triple A	336
10.1.5	Die Matcher von Vitest	338
10.1.6	Gruppierung von Tests – Testsuites	339
10.1.7	Setup- und Teardown-Routinen	340
10.1.8	Tests überspringen und exklusiv ausführen	341
10.1.9	Umgang mit Exceptions	343
10.1.10	Testen von asynchronen Operationen	345
10.2	Testen von Hilfsfunktionen	347
10.3	Snapshot-Testing	348
10.3.1	Snapshot-Tests für Komponenten	349
10.4	Komponenten testen	353
10.4.1	Test der »BooksListItem«-Komponente	354

10.4.2	Interaktion testen	357
10.4.3	Formulare testen	358
10.5	Umgang mit Serverabhängigkeiten	361
10.5.1	Fehler bei der Kommunikation simulieren	364
10.6	Ende-zu-Ende-Tests	366
10.6.1	Tests mit dem Vitest Browser Mode	366
10.7	Zusammenfassung	368
11	Komponentenbibliotheken in einer React-Applikation	369
11.1	Installation und Integration von Material-UI	369
11.2	Listendarstellung mit der »Table«-Komponente	371
11.2.1	Die Liste in der Tabelle filtern	373
11.3	Grids und Breakpoints	376
11.4	Icons	378
11.5	Einen Bestätigungsdialog implementieren	381
11.6	Formulare mit Material-UI	386
11.7	shadcn/ui als Alternative zu Material-UI	390
11.7.1	Installation von shadcn/ui	390
11.7.2	shadcn/ui-Komponenten nutzen	391
11.8	Zusammenfassung	392
12	Navigation innerhalb einer Applikation – der Router	393
12.1	Installation und Einbindung	394
12.1.1	Die verschiedenen Router	395
12.2	Navigation in der Applikation	396
12.2.1	Es wird immer die beste Route aktiviert	398
12.2.2	Navigation zwischen verschiedenen Routen	398
12.2.3	Layout-Komponenten im React Router	400
12.3	»Not found«	402
12.4	Testen des Routings	404
12.5	Bedingte Umleitungen	408
12.6	Dynamische Routen	411

12.7	Imperative Navigation	413
12.8	Routing mit dem TanStack Router	414
12.8.1	Installation und Konfiguration	415
12.8.2	Routen definieren	417
12.9	Zusammenfassung	421
13	Eigene React-Bibliotheken erzeugen	423
13.1	Eine eigene Komponentenbibliothek erzeugen	423
13.1.1	Initialisierung der Bibliothek	424
13.1.2	Die Struktur der Bibliothek	428
13.1.3	Hooks in der Bibliothek	429
13.1.4	Das Bauen der Bibliothek	430
13.2	Einbinden der Bibliothek	431
13.2.1	Reguläre Installation des Pakets	433
13.3	Testen der Bibliothek	433
13.3.1	Unit-Test für die Bibliothekskomponente	434
13.3.2	Unit-Test des Custom Hooks der Bibliothek	434
13.4	Storybook	436
13.4.1	Installation und Konfiguration von Storybook	436
13.4.2	Die Button-Story in Storybook	437
13.5	Zusammenfassung	439
14	Zentrales State-Management mit Redux	441
14.1	Die Flux-Architektur	442
14.1.1	Der zentrale Datenspeicher – der Store	442
14.1.2	Die Anzeige der Daten mit den Views	443
14.1.3	Actions – die Beschreibung von Änderungen	443
14.1.4	Der Dispatcher – die Schnittstelle zwischen Actions und dem Store	444
14.2	Installation von Redux	445
14.2.1	Die Struktur der Applikation	446
14.3	Den zentralen Store konfigurieren	446
14.3.1	Debugging mit den Redux Dev Tools	448
14.4	Der Umgang mit Änderungen am Store mit Reducern	449
14.4.1	Der »Books«-Slice	449
14.4.2	Den »BooksSlice« einbinden	452

14.5	Komponenten und den Store verknüpfen	453
14.5.1	Anzeige der Daten aus dem Store	453
14.5.2	Selektoren	455
14.6	Änderungen mit Actions beschreiben	457
14.6.1	Löschen von Datensätzen	458
14.7	Datensätze erstellen und bearbeiten	461
14.8	Zusammenfassung	466
15	Umgang mit Asynchronität und Seiteneffekten in Redux	469
15.1	Middleware in Redux	469
15.1.1	Eine eigene Middleware implementieren	470
15.2	Redux mit Redux Thunk	471
15.2.1	Manuelle Integration von Redux Thunk	472
15.2.2	Daten vom Server lesen	472
15.2.3	Datensätze löschen	480
15.2.4	Datensätze anlegen und modifizieren	484
15.3	Zusammenfassung	488
16	Serverkommunikation mit GraphQL und dem Apollo-Client	491
16.1	Einführung in GraphQL	491
16.1.1	Die Charakteristik von GraphQL	491
16.1.2	Die Nachteile von GraphQL	492
16.1.3	Die Prinzipien von GraphQL	493
16.2	Apollo, ein GraphQL-Client für React	497
16.2.1	Installation und Einbindung in die Applikation	497
16.2.2	Lesender Zugriff auf den GraphQL-Server	499
16.2.3	Zustände einer Anfrage	501
16.2.4	Löschen von Datensätzen	503
16.3	Die Apollo Client Devtools	505
16.4	Lokales State-Management mit Apollo	506
16.4.1	Den lokalen State initialisieren	507
16.4.2	Den lokalen State benutzen	508
16.5	Zusammenfassung	511

17 Internationalisierung	513
17.1 Einsatz von react-i18next	514
17.1.1 Sprachdateien vom Backend laden	518
17.1.2 Die Sprache des Browsers verwenden	519
17.1.3 Die Navigation um eine Sprachumschaltung erweitern	520
17.2 Platzhalter verwenden	522
17.3 Werte formatieren	525
17.3.1 Zahlen und Währungen formatieren	525
17.3.2 Datumswerte formatieren	527
17.4 Singular und Plural	529
17.5 Zusammenfassung	532
18 Performance	533
18.1 Der Callback-Hook	533
18.2 React.memo	537
18.3 Der React Compiler	540
18.3.1 Voraussetzungen für den React Compiler	540
18.3.2 Installation des Compilers	541
18.3.3 Optimierungen mit dem React Compiler	541
18.4 Rules of React	543
18.5 »React.lazy« – Suspense for Code Splitting	544
18.5.1 Lazy Loading in einer Applikation	545
18.5.2 Lazy Loading mit dem React Router	549
18.6 Suspense for Data Fetching	552
18.7 Virtuelle Tabellen	553
18.8 Zusammenfassung	558
19 Authentifizierung in einer React-Applikation	561
19.1 Grundlagen tokenbasierter Authentifizierung	562
19.1.1 JWT – JSON Web Token	562
19.1.2 Arten von Tokens	563
19.1.3 OpenID Connect (OIDC)	563
19.2 Authentifizierungs-State und Token-Handling in React	564
19.2.1 Setup des Identitätsproviders	564
19.2.2 Authentifizierung in der React-Applikation	567

19.3	Geschützte Ressourcen und Requests	569
19.3.1	Implementierung des Backends	569
19.3.2	Kommunikation mit dem Backend	570
19.4	Arbeiten mit Rollen	573
19.5	Zusammenfassung	575
20	Progressive Web Apps	577
20.1	Merkmale einer Progressive Web App	577
20.2	Initialisieren der Applikation	578
20.3	Installierbarkeit	580
20.3.1	Die sichere Auslieferung einer Applikation	580
20.3.2	Das Web-App-Manifest	581
20.3.3	Service-Worker in der React-Applikation	584
20.3.4	Installation der Applikation	584
20.3.5	Die Benutzer fragen	586
20.4	Offlinefähigkeit	590
20.4.1	Eigene Service-Worker	590
20.4.2	Umgang mit dynamischen Daten	594
20.4.3	Offlineunterstützung im Service-Worker	600
20.5	Zusammenfassung	602
21	Native Apps mit React Native	605
21.1	Der Aufbau von React Native	605
21.2	Die Installation von React Native	606
21.2.1	Die Projektstruktur	606
21.2.2	Die Applikation starten	607
21.3	Anzeige einer Übersichtsliste	610
21.3.1	Statische Listenansicht	611
21.3.2	Styling in React Native	614
21.3.3	Suchfeld für die »List«-Komponente	620
21.3.4	Serverkommunikation	622
21.4	Debugging in der simulierten React-Native-Umgebung	624
21.5	Bearbeiten von Datensätzen	625
21.5.1	Implementierung der »Form«-Komponente	626
21.6	Publizieren	633
21.6.1	Build der App	634
21.7	Zusammenfassung	635

22	Next.js – Fullstack-React – Grundlagen	637
22.1	Next.js – die Hintergründe	638
22.2	Installation	638
22.2.1	Die Applikation starten	641
22.3	Die Struktur einer Next.js-Applikation	641
22.4	React Server Components	643
22.4.1	Implementierung einer Server Component	644
22.4.2	Fehlerbehandlung in Server Components	647
22.5	Statisches vs. dynamisches Rendern	648
22.5.1	Wann rendert Next.js dynamisch?	648
22.6	Statische Generierung	651
22.7	Der App Router	652
22.7.1	Layouts	652
22.7.2	Dynamische Routensegmente	653
22.7.3	Dynamische Routensegmente statisch rendern	655
22.7.4	Umleitungen	657
22.7.5	Route Groups	659
22.7.6	Parallel Routes	662
22.7.7	Intercepting Routes	663
22.7.8	Route Handlers	669
22.8	Proxy	672
22.9	Zusammenfassung	673
23	Verbesserung der User-Experience mit Next.js	675
23.1	Client Components in Next.js	675
23.1.1	Verschachtelung von Client und Server Components	680
23.2	Arbeiten mit Suchparametern	680
23.2.1	Suchparameter in Server Components	681
23.2.2	Suchparameter in Client Components	683
23.3	»loading.tsx« – Ladezustände abbilden	685
23.4	»error.tsx« – Fehler abfangen	688
23.5	»not-found.tsx« – Anzeige bei fehlenden Daten	692
23.6	Caching in einer Next.js-Applikation	694
23.6.1	»fetch«-Cache	694
23.6.2	Full Route Cache	697
23.6.3	Router Cache	698

23.7	Revalidierung und Datenaktualisierung	698
23.7.1	Zeitgesteuerte Invalidierung	699
23.7.2	Pfadbasierte Invalidierung	700
23.7.3	Tag-basierte Invalidierung	703
23.8	Streaming	704
23.9	Cache Components	707
23.9.1	Die neuen APIs der Cache Components	709
23.9.2	Statische und dynamische Inhalte in einer Cache Component	711
23.10	Zusammenfassung	714
24	Server Functions in Next.js	717
24.1	Server Functions definieren und ausführen	717
24.2	Server Functions an Client Components weitergeben	720
24.3	Server Functions in Formularen	722
24.4	Hooks für die Arbeit mit Formularen	724
24.4.1	Den Pending-Zustand mit »useFormStatus« abdecken	724
24.4.2	Formulare mit dem »useActionState« absenden	726
24.5	Das Zusammenspiel von Server Functions und React Hook Form	731
24.6	Mit Transitionen arbeiten	734
24.7	Die »use()«-Funktion in Next.js	736
24.8	Zusammenfassung	738
25	Weitere Optimierungen in Next.js	739
25.1	Umgang mit Metadaten	739
25.1.1	Statische Metadaten	740
25.1.2	Dynamische Metadaten	742
25.1.3	Favicon, Open Graph Images und weitere dateibasierte Metadaten	743
25.2	Optimierung von Schriftdateien	746
25.3	Bilder optimieren	747
25.4	Prefetching von Links	750
25.5	Zusammenfassung	752

26 Künstliche Intelligenz in React-Applikationen	753
26.1 Architektur einer KI-Applikation	754
26.2 Modell- und Backend-Setup	756
26.3 Kommunikation zwischen React und dem LLM	757
26.3.1 Kommunikation mit einem LLM	757
26.3.2 Streaming von KI-Kommunikation in React	761
26.4 Tools in React-Applikationen aufrufen	764
26.5 Personalisierung von UIs mit KI	769
26.6 Zusammenfassung	773
Index	775

Index

A

Access-Token	563
Action (JavaScript)	457
Action Creator	444
Action-Objekt	265
<i>payload</i>	265
<i>type</i>	265
Airbnb-Style-Guide	89
Aktualisierung	66
<i>automatische</i>	67
Alternativen	35
Anforderungen	
<i>Browser</i>	60
<i>Editor</i>	59
<i>Node.js</i>	59
Angular	29
Anmeldeformular	224
Anonyme Funktion	136
API-Funktionen	197
Apollo	497
<i>@apollo/react-hooks</i>	498
<i>apollo-boost</i>	498
<i>ApolloClient</i>	499
<i>ApolloProvider</i>	499
<i>Chrome-Extension</i>	505
<i>Client</i>	491
<i>Context</i>	498
<i>graphql</i>	498
<i>Initialisierung</i>	507
<i>InMemoryCache</i>	499
<i>Installation</i>	497
<i>lesen</i>	499
<i>Listendarstellung</i>	499
<i>Local Resolver</i>	507
<i>local-only Fields</i>	507
<i>lokaler State</i>	508
<i>löschen</i>	503
<i>makeVar</i>	507
<i>Mutation-Hook</i>	503
<i>Reactive-Variablen</i>	507
<i>refetchQueries</i>	503
<i>State-Management</i>	506
<i>useReactiveVar</i>	510
<i>Zustände</i>	501
Apollo Client Devtools	505
<i>Cache</i>	506
<i>Explorer</i>	506
<i>Mutations</i>	506
<i>Queries</i>	506
App Router	652
App.jsx	83
Applikationslogik	168
ARIA	383
Arrow-Funktion	89
Asset, statisches	74
async/await	164
Asynchrone Operation	162
Aufbau	46
Authentifizierung	561
<i>tokenbasierte</i>	562
Autofocus	225
autofocus (Attribut)	225
Autorisierung	561
Axios	206
B	
Babel	52, 55
<i>JSX</i>	56
<i>text/babel</i>	56
Backend-Implementierung	244
Basis-Hooks	260, 297
beforeinstallprompt	586
Benannter Export	89
Benutzeroberfläche	36
Bestätigungsdialog	381
Bibliothek	423
<i>Build-Prozess</i>	430
<i>einbinden</i>	431
<i>Export</i>	429
<i>Hooks</i>	429
<i>Hook-Test</i>	434
<i>initialisieren</i>	424
<i>Komponententest</i>	434
<i>Struktur</i>	428
<i>testen</i>	433
<i>Verzeichnisstruktur</i>	428
Bibliotheks-Hooks	295
Biome	640
Bootstrap	369
Breaking Change	36, 67
Breakpoint	75
Browsercache	29
Browsererweiterung	76
Browserfeatures	577

BrowserRouter 395
 Build
 JSX Transform 88
 Übersetzung 88
 Build-Prozess 58, 77, 393, 483
 Verzeichnis 77
 Webserver 77

C

Callback-Hook 271
 CDN 370
 Change-Event 248
 children-Prop 180
 Chrome 59–60
 className 305
 classNames 307
 Cleanup 156
 clearInterval 155
 Client Components 644
 closest 120
 Codebeispiele 47
 Codemod 36
 CodePen 51
 Login 51
 React 52
 Codestyle 85
 Concurrency Patterns 552
 Concurrent Mode 33
 Concurrent Renderer 34
 const 135
 Container Component 167
 einbinden 170
 Logik auslagern 168
 Content Delivery Network → CDN
 Context 453
 Context-API 33, 184
 contextType 33
 Controlled Component 221, 234
 synchronisieren 235
 Cordova 605
 Create React App 49
 npx 61
 Optionen 65
 TypeScript 50
 yarn create 65
 createElement 94
 createRoot 82
 cross-env 166
 CRUD 48
 CSS 91, 302
 Basis-Selektoren 304

CamelCase 313
Eigenschaften 302
Einheit 313
global 91
importieren 91, 301
Klassennamen 305
Namespacing 303
Regeln 313
Selektoren 302
Spezifität 306
Wurzelement 303
 CSS Flexbox 376
 CSS-Framework 325
 css-in-js 316
 css-in-js-Bibliotheken 295
 CSS-Modul 313, 660
 Benennungsschema 315
 Dateiendung 314
 Klassennamen 315
 Pseudoselektoren 315
 standardkonformes 314
 CSS-Präprozessor 309
 Custom Hook 247, 296, 586

D

Data Table 529
 data-testid 355
 Dateiendung 80
 Dateiname 87
 Dateiupload 240
 Uncontrolled Component 240
 Datenfluss 40, 47
 Datenklasse 95
 Datensatz 201
 löschen 201
 validieren 207
 verändern 203
 vom Server laden 211
 Datenstrom 40
 Debugger 74
 Navigation 75
 Visual Studio Code 76
 WebStorm 76
 Debugging 54
 DebugValue-Hook 282
 Defaultexport 89
 DeferredValue-Hook 284
 DefinitelyTyped 134
 Deprecation 36
 describe.each 405
 Destructuring 53

devDependency 134
 Dexie 597
 Installation 597
 Dialog, open-Prop 381
 Dispatch 265–266
 Docker 564
 Container 564
 DOM 42
 Dumb Component 167

E

E2E-Test 334
 ECMAScript 129
 ECMAScript-Modulsystem 69
 Edge 60
 EffectEvent-Hook 290
 Effect-Hook 146
 Eindeutige Identifier 289
 Einstiegshürde 260
 Element 37, 93
 Attribute 93
 immutable 94
 Eltern-Kind-Beziehung 40
 Emotion 316
 Autocompletion 320
 css-Prop 316–317
 css-Template-String 319
 externes Objekt 319
 Inline-Objekt 318
 Installation 316
 styled-Ansatz 316, 319
 Syntax-Highlighting 320
 Emotion Styled Components 319
 Ende-zu-Ende-Test 334, 366
 Entwicklerwerkzeuge 75
 Entwicklung
 lokale 54
 testgetriebene 335
 Entwicklungsprozess 58
 Entwurfsmuster 173, 300
 ErrorBoundary 213
 error-Prop 230
 ESLint 82, 85, 298, 640
 Fehler 86
 esm.sh 53
 Event-Behandlung 119
 Event-Binding 111
 Event-Handler 41, 116
 Event-Objekt 117
 Event-Pooling 117

Events 111
 Callback 112
 Klick 111
 Exception 76
 Expo 606
 Expo Terminal UI 608
 Export 89
 benannter 89

F

Facebook 27
 Fassade 494
 FaxJS 30
 Fehlersuche 74
 Fetch on Render 552
 Fetch then Render 553
 Fetch-API 160
 Fiber 32, 259
 files-Eigenschaft 243
 Firefox 60
 Flexbox 376
 Flow 128
 Flux 45, 262, 442
 Action 443
 Darstellung 443
 Dispatcher 444
 Store 442
 View 443
 Flux Standard Action 443
 Kriterien 443
 payload 444
 type 443
 FormData 243
 Formular 221
 absenden 226
 Styling 256
 ForwardRef 277–278
 Fragment 192
 Funktionskomponente 40, 84, 259

G

Generics 138
 GitHub 28
 Git-Repository 73
 Installation 433
 Globales Stylesheet 303
 Google Font 746
 Gradual Upgrade 34

GraphiQL 496, 505
Autovervollständigung 496
Fehler-Highlighting 496
GraphQL 491
Abfragesprache 492
Applikationslogik 496
Caching 493
Datenstruktur 493
Datentypen 494
deprecated 492
Dokumentation 492
Filter 495
gql 500
Graphen 492
Mutations 496
Nachteile 492
Overhead 493
Plain Text 493
Query 495
Resolver 494
Schema 493
Struktur 492
Typsistem 492
Versionierung 492
Grid-System 376
Grundlagen 79

H

Hash-Navigation 395
HashRouter 395
HATEOAS 200
Higher-Order Component 173, 537
Beispiel 173
einbinden 176
with-Präfix 175
History-API 45, 395
HOC → Higher-Order Component
Hook
Auslagerung 296
Callback 533
Context-Hook 261
createRef 222
Effect-Hook 260
Entwurfsmuster 260
Funktionskomponenten 299
kleine Komponenten 259
Lifecycle 260
Memo 537
Namenskonvention 260
Platzierung 299
Regeln 298

Reihenfolge 298
State 260
Top Level 298
weniger Duplikate 259
Hook-API 33, 259
HTMLInputElement 224
HTML-Referenzen 275
HTTP 199
DELETE 200–201
GET 200
PATCH 200
POST 200, 203
PUT 200
HTTP-Methode 669
http-server 56, 77
globale Installation 57
lokale Installation 56
On-Demand-Ausführung 57
Hydratation-Prozess 289

I

i18N 513
Datum 513
Mehrsprachigkeit 513
Zahlen 513
i18next 514
changeLanguage 521
count 529
Datum 527
Fallback-Sprache 515
i18next-http-backend 518
init 515
Konfiguration 514
language 521
Pluralregeln 530
Ressourcendateien 515
Sprachdateien 518
Währung 527
IconButton 379
Identitätsprovider 564
Id-Hook 289
ID-Token 563
IIFE 164
Immer 123, 459
draftState 124
produce 123
Immutability 121
immutability-helper 123
immutable 459
Immutable Data Structures 122
Immutable.js 123

ImperativeHandle-Hook	280
Import	89
<i>gruppieren</i>	90
IndexedDB	597
Initiales Rendern	152
Initialisierung	79
Ink	42
Inline-Styling	311
InsertionEffect-Hook	295
Interaktionsmöglichkeit	379
Internationalisierung	513
Intl-Schnittstelle	525
isfiberreadyet	32

J

Jasmine	46, 331
Jest	331
<i>Seiteneffekt</i>	347
JSCodeshift	67
jsdom	331
JSON	162
JSON Web Token	562
json-server	158
JSX	37, 80, 82, 92, 305
<i>AND-Operator</i>	101
<i>Ausdrücke</i>	95
<i>Bedingungen</i>	100
<i>dangerouslySetInnerHTML</i>	98
<i>Erweiterbarkeit</i>	98
<i>Exception</i>	97
<i>if-Statements</i>	101
<i>Iterationen</i>	98
<i>JavaScript-Ausdrücke</i>	93
<i>JavaScript-Funktionalität</i>	98
<i>Kommentar</i>	97
<i>map</i>	98
<i>Schleifen</i>	98
<i>Schutzmechanismus</i>	97
<i>statische Werte</i>	95
<i>Syntaxerweiterung</i>	94
<i>Ternäroperator</i>	102
<i>undefined</i>	97
JWT → JSON Web Token	

K

Karma	46
key-Attribut	99
Keycloak	564

key-Prop	42, 44
<i>Zuordnung</i>	44

KI

<i>Architektur</i>	754
<i>Claude</i>	755
<i>Container</i>	756
<i>dangerouslyAllowBrowser</i>	757
<i>GPT</i>	755
<i>große Sprachmodelle</i>	756
<i>Kontext</i>	761
<i>Llama 3.2</i>	756
<i>Llama.cpp</i>	755
<i>LLM</i>	756
<i>Ollama</i>	755–756
<i>OpenAI</i>	757
<i>OpenAI-API</i>	757
<i>Personalisierung</i>	769
<i>Streaming</i>	761
<i>Token</i>	762
<i>Tool-Calling</i>	764
<i>Tool-Definition</i>	765
<i>Vektordatenbank</i>	755

KI-Features	753
Kindelement	40
Kindkomponente	47, 104
Klassenkomponente	39
<i>State</i>	109
Klassenname	
<i>bedingter</i>	308
<i>mehrere</i>	307
<i>zusammenfügen</i>	308
Kommandozeile	605
Kommandozeilenwerkzeug	58
Komponente	37, 79, 145
<i>Aufteilung</i>	169
<i>aushängen</i>	156
<i>Dateiendung</i>	87, 223
<i>Großbuchstaben</i>	93
<i>Lebenszyklus</i>	145
<i>Tag</i>	106
Komponentenbaum	38, 79, 193, 441
Komponentenbibliothek	369, 423
Komponentenfunktion	95
Komponentenhierarchie	46
Komponentensammlung	45
Komposition	260
Komprimierer	544
Kontext	184
<i>Beispiel</i>	188
<i>createContext</i>	184
<i>einbinden</i>	186
<i>erzeugen</i>	185

Kontext (Forts.)
Kindelemente 189
Zugriff 187
 Ky 206
installieren 206

L

LayoutEffect-Hook 281
 lazy (Funktion) 33
 Lazy Loading 29, 545
 Lebenszyklus 145
 Komponente aktualisieren 150
 Komponente einhängen 146
 Mount 145–146
 Unmount 145, 154
 Update 145, 150
 Lesbarkeit 271
 let 135
 Lifecycle 40
 Lifecycle-Methode 33
 Link
 symbolischer 431
 Linux Foundation 491
 Loading-Indicator 149
 Logik 167
 Lokale Entwicklung 54

M

main.jsx 80
 Mangler 544
 manifest.json 74
 Material Design 45, 369
 Material-UI 45, 113, 369
 Breakpoint 376
 Controlled Component 374
 DialogActions 383
 DialogContent 383
 DialogTitle 383
 Filter 373
 Formulare anlegen 386
 Grid 376
 Icons 370, 378
 installieren 369
 Scrollbar 378
 spacing 376
 Table 371
 Memo-Hook 273
 Memoisierung 261–262, 271
 Meta 27

Metro-Bundler 608
 Middleware 469
 MIT-Lizenz 28
 Mixin 309
 Mobile Endgeräte 605
 Mocha 331
 Mock 46
 Mock Service Worker 361–362
 Modulsystem 89
 Multi-Faktor-Authentifizierung 564
 Multi-Page-Applikation 29
 Mutation 211, 215
 Mutation-Typ 496

N

Native App 605
 Navigation 45, 393
 Nest.js 669
 Next.js 34, 637
 action 722
 App Router 640
 async 646
 Barrierefreiheit 739
 Bilder optimieren 747
 Build-Ausgabe 650
 Build-Optimierungen 695
 Cache 642, 694, 707
 Cache Components 707
 Cache Components, Unterschiede 714
 Cache, auto no cache 697
 Cache, deaktivieren 696
 Cache, expire 710
 Cache, force-cache 697
 Cache, no-store 697
 Cache, Profile 709
 Cache, revalidate 710
 Cache, stale 710
 Cache, tag-basierter 695
 Cache, zeitgesteuerter 695
 cacheLife 709
 cacheTag 711
 charset 740
 Client Component 675
 Client Components rendern 679
 Controlled Component 731
 create-next-app 638
 CSS-Postprozessor 643
 dispatchAction 726
 dynamische Funktion 649
 dynamische Metadaten 742
 dynamische Routensegmente 653

Next.js (Forts.)	
dynamisches Rendern	648
error.tsx	688
ErrorBoundary	688
Favicon	743
Fehlerbehandlung	647, 688
fetch-Cache	694
force-dynamic	680
FormData	722
formData.entries()	724
Formularbibliothek	731
Full Route Cache	697
generateMetadata()	742
generateStaticParams()	655
global-error.tsx	691
Grundlagen	637
Hot Module Reload	641
Hydration	679, 697
Image	747
ImageResponse	744
Import Alias	640
Installation	638
Intercepting Routes	663
Invalidierung	729
Invalidierung, pfadbasierte	698, 700
Invalidierung, tag-basierte	698, 703
Invalidierung, zeitbasierte	698
Invalidierung, zeitgesteuerte	699
Komponentenarten	680
Konfiguration	642
Ladezustand	685
Layout	652
Layout, Verschachtelung	652
Layout-Komponente	652
Lazy Loading	749
Link	751
Linter	640
Linter-Konfiguration	642
loading.tsx	685
localFont()	747
metadata	740
Metadaten	739
Namenskonflikte	661
next.config.ts	657
next.revalidate	699
NextPage	646
not-found.tsx	692
notFound()	692
Object.fromEntries()	724
OG Images	744
Open Graph Images	743
page.tsx	646
Page-Komponente	642, 645
Pages Router	640
Parallel Routes	662
params-Prop	654
Pending-Zustand	724
permalink	726
Prefetching	698, 750–751
Proxy	672
Query-Parameter	681
React Hook Form	731
React Server Component Payload	697
Reconciliation	697
reducerAction	726
Renderfehler	648
Renderzyklus	644
Request-Deduplication	695
reset()	690
revalidate	695
revalidatePath()	702, 718
revalidateTag()	703
Revalidierung	698
robots.txt	744
Root-Layout	642, 652, 741
Route Groups	659
Route Handler	677
Route Handlers	669
Route Handlers, dynamische	671
route.ts	669
router.push()	701
RSC	697
RSC-Payload	719, 752
Runtime	643
Schriftdateien	746
searchParams-Prop	682
SEO	739
Server Components	643
Server Function	717
Server Function, Endpunkt	717
Server Function, Formulare	722
Server Function, Serialisierung	717
Server Function, Weitergabe	720
sitemap.xml	745
Slot	662
starten	641
startTransition()	735
statische Dateien	642
statische Generierung	651
statische Metadaten	740
statisches Rendern	648, 655
Status 404	692
Streaming	704
Suchmaschinenoptimierung	739

Next.js (Forts.)

- Suchparameter* 649, 680
- Suspense* 685, 704
- Tailwind* 640
- Transitionen* 734
- try-catch* 647
- TypeScript* 640
- Umleitung* 657
- Umleitung, dynamische* 658
- Umleitung, statische* 657
- updateTag* 711
- URLSearchParams* 684
- use cache* 708
- use client* 676
- use server* 719
- use()* 736
- useActionState()* 724, 735
- useFormStatus()* 724
- useRouter()* 684
- useSearchParams()* 684
- Validierung* 734
- Validierungsschema* 727
- variabler Pfad* 653
- Verschachtelung* 680
- View Transitions* 734
- viewport* 740
- WebP* 749
- Zeichencodierung* 740
- Zwischenspeicher* 694

Ngix 395

Niedrige Priorität 287

No New Features 34

node_modules 64, 74

Node.js 59

NPM 59, 62, 424

npm link 431

npm outdated 66

npm run build 77

npm run dev 84

NPM-Kommandos 63

NPM-Paket 431

NPM-Registry 433

nth-child 322

nth-of-type 322

O

- OAuth 2.0 564
- Objektstruktur 121
- tiefe* 121
- Offlinefähigkeit 577, 590
- Open Graph Images 744

- Open Source 27
- OpenID Connect (OIDC) 563
- Optimierung 271
- Optimistic-Hook 292

P

- package.json 62, 73
- scripts* 69
- package-lock.json 73
- Paginierung 554
- Paket

 - installieren* 433
 - veröffentlichen* 433

- Paketmanager 59, 63
- Parameter 104
- Parameter Properties 138
- Parcel 67
- PascalCase 87
- Performance 32, 533

 - Bundle-Größe* 544
 - isEqual* 540
 - Lesbarkeit* 533
 - Optimierung* 533

- Permanent Redirect 657
- Plattformunabhängigkeit 33
- Playground 51
- Playwright 366
- Polyfill 60
- Präprozessor 309
- Presentational Component 167

 - implementieren* 171

- preventDefault 227
- Produktivapplikation 167
- Progressive Web App → PWA
- Promise 164
- Prop 38, 41, 103
- PropTypes 106
- Pseudoselektor 311
- public-Verzeichnis 74
- Pure Functions 175
- PWA 577

 - Anpassungsfähigkeit* 578
 - App-Switcher* 584
 - Auffindbarkeit* 578
 - Benachrichtigung* 578
 - Cache leeren* 590
 - Checkliste* 578
 - display* 585
 - dynamische Daten* 594
 - HTTPS* 580
 - Installation* 584

- PWA (Forts.)
- Installation-Prompt* 586
 - Installationsprozess* 584
 - Installationsvoraussetzungen* 580
 - Installierbarkeit* 578, 580
 - Integration* 578
 - Konfliktlösung* 594
 - Konfliktstrategien* 594
 - lokales Speichern* 597
 - Merkmale* 577
 - Offlinefähigkeit* 578
 - Offlinemodus* 599
 - online-Event* 597
 - Service-Worker* 584
 - Service-Worker deaktiviert* 580
 - sicherer Kontext* 580
 - Sicherheit* 578
 - standalone* 585
 - Startleiste* 584
 - Synchronisierung* 597
 - Unabhängigkeit* 577
 - Web-App-Manifest* 581
 - Zuverlässigkeit* 578
- Q**
-
- Quellcode
- Organisation* 222
 - Query* 211
 - querySelector* 222
 - Query-Typ* 494
- R**
-
- React
- Authentifizierung* 564
 - Context* 441
 - Element* 93
 - in HTML einbinden* 54
 - Referenzen* 222
 - Server Components* 643
 - Serverkommunikation* 195
 - React (Konzepte)* 37
 - React Compiler* 65, 540, 640
 - React Developer Tools* 76, 282
 - Components* 76
 - Profiler* 76
 - React Hook Form* 221, 247, 358, 386, 462
 - Daten laden* 249
 - Fehler* 251
 - Fehlermeldung* 256
 - formState* 251
 - handleSubmit* 248
 - register* 247
 - Resolver* 256
 - Validierung* 251
 - Validierungsregeln* 253
 - Validierungsschema* 254
 - React Native* 42, 605
 - Android* 605
 - Android-Build* 634
 - app.json* 633
 - Apple App Store* 605, 633
 - Aufbau* 605
 - Außenabstand* 616
 - autoCapitalize* 621
 - automatischer Reload* 612
 - Axios* 622
 - Build* 633-634
 - Controlled Component* 620
 - create* 615
 - CSS-Syntax* 615
 - Debugging* 624
 - editieren* 625
 - Eingabefeld* 620
 - Emotion* 618
 - Expo-Go-App* 610
 - Fehlermeldung* 612
 - Fetch-API* 622
 - FlatList-Komponente* 613
 - Flex-Layout* 617
 - Formular* 625
 - Gerät* 610
 - Google Play Store* 605, 633
 - Installation* 606
 - iOS* 605
 - iOS-Build* 634
 - Java* 605
 - JavaScriptCore* 605
 - keyExtractor-Prop* 614
 - Kotlin* 605
 - native GUI-Elemente* 611
 - Objective-C* 605
 - placeholder-Prop* 621
 - Projektstruktur* 606
 - publizieren* 633
 - renderItem-Prop* 614
 - Routen* 625
 - Routenparameter* 626
 - Serverkommunikation* 622
 - Simulator* 606, 609
 - Start* 607

React Native (Forts.)	
<i>style-Props</i>	619
<i>StyleSheet</i>	615
<i>Styling</i>	614
<i>Swift</i>	605
<i>Tastaturkürzel</i>	624
<i>TextInput-Komponente</i>	620
<i>Text-Komponente</i>	612
<i>Trennelement</i>	617
<i>TypeScript</i>	613
<i>View-Komponente</i>	612
<i>Webview</i>	605
<i>Worker-Prozess</i>	625
<i>Xcode</i>	606
<i>XMLHttpRequest-API</i>	622
React Query → TanStack Query	
React Router	393, 462
<i>bedingte Umleitungen</i>	408
<i>dynamische Routen</i>	411
<i>element</i>	397
<i>Fehlerseite</i>	402
<i>initialEntries</i>	407
<i>installieren</i>	394
<i>Layout-Komponenten</i>	400
<i>Lazy Loading</i>	549
<i>Modes</i>	394
<i>Navigate-Komponente</i>	397
<i>not found</i>	402
<i>path</i>	397
<i>Priorität</i>	398
<i>Test</i>	404
<i>Variable</i>	398
React Scripts	
<i>start</i>	68
React Testing Library	349, 354, 358, 367, 424, 435
React.FC	141
React.Fragment	193
<i>leeres Tag</i>	193
React.lazy	544
React.memo	537
React-Applikation	
<i>Anforderungen</i>	59
react-codemod	67
react-dom	42, 725
react-error-boundary	213
react-i18next	513
<i>Browsersprache</i>	519
<i>dynamische Werte</i>	522
<i>Installation</i>	514
<i>Platzhalter</i>	522
<i>Singular und Plural</i>	529
<i>Sprachumschaltung</i>	520
<i>t-Funktion</i>	518
<i>useTranslation</i>	515
<i>Zahlenformat</i>	525
react-redux	445
Readme	73
Realm	566
Reconciler	28, 32, 42
<i>Algorithmus</i>	42
<i>Annahmen</i>	42
<i>Optimierung</i>	43
Reducer	45, 444
Reducer-Funktion	265
Reducer-Hook	263
<i>Asynchronität</i>	266
<i>Middleware</i>	266
Redux	45, 295, 441, 444
<i>Action</i>	457
<i>Action Creator</i>	472
<i>Action fulfilled</i>	475
<i>Action pending</i>	475
<i>Action rejected</i>	475
<i>Action-Typ</i>	457
<i>Aggregation</i>	456
<i>applyMiddleware</i>	470
<i>configureStore</i>	446
<i>Context</i>	447
<i>createSlice</i>	451–452
<i>Daten laden</i>	478
<i>Datensatz bearbeiten</i>	461
<i>Datensatz erstellen</i>	461
<i>Debugging</i>	448
<i>dispatch</i>	457
<i>Fehlerbehandlung</i>	479, 484
<i>Higher-Order-Selektor</i>	456
<i>initialState</i>	452
<i>Installation</i>	445
<i>loadingState</i>	477
<i>löschen</i>	458, 480
<i>Middleware</i>	448, 469
<i>Overhead</i>	445
<i>PayloadAction</i>	459
<i>Reducer</i>	444, 449
<i>RootState</i>	447
<i>Seiteneffekte</i>	449, 469
<i>Selektoren</i>	455
<i>Serverkommunikation</i>	449
<i>Slice</i>	446, 449
<i>Slice einbinden</i>	452
<i>Store</i>	446
<i>Store-Zugriff</i>	453
<i>striktter Workflow</i>	441

- Redux (Forts.)
 - Struktur* 446
 - Teilstate* 449
 - useAppDispatch* 458
 - useSelector* 454
 - Redux Dev Tools 448, 470
 - Redux Saga 488
 - Redux Thunk 447, 470–471
 - Alternativen* 488
 - Daten lesen* 474
 - erzeugen* 484
 - Integration* 472
 - modifizieren* 484
 - Serverkommunikation* 472
 - Redux Toolkit 445
 - Redux-Middleware
 - Action* 471
 - Implementierung* 470
 - Next* 471
 - Store* 470
 - Redux-Observable 488
 - Ref 222, 262
 - current* 225–226
 - Referenz 222
 - Referenzierung 99
 - Ref-Hook 275
 - Formular* 275
 - Werte speichern* 276
 - Refresh-Token 563
 - Regeln, Datei 87
 - Relay 497
 - Release 29
 - Release-Zyklus 36
 - Render as you fetch 553
 - Render Prop 178
 - Benennung* 180
 - Integration* 182
 - render() 37
 - Renderer 28, 42
 - Rendering 80
 - render-Methode 533
 - Render-Priorität 283
 - Renderprozess 146
 - renderToReadableStream 637
 - Render-Vorgang 116
 - Request for Comments (RFC) 31
 - Ressourcen 154
 - REST (Representational State Transfer) 200, 491
 - REST-Schnittstelle 200
 - Roboto 370, 746
 - Robots Exclusion Standard 744
 - Rollen 573
 - Root-Objekt 53
 - Router 45
 - Rsbuidl 68
 - Rules of Hooks 298, 736
- ## S
-
- Safari 60
 - Sass 309
 - Mixins* 309
 - Variablen* 309
 - Verschachtelung* 309
 - Schema 208
 - Scrollen, virtuelles 554
 - SCSS 309
 - @import* 310
 - Farbvariablen* 310
 - Verschachtelung* 310
 - Seiteneffekt 146, 150
 - Seitenwechsel 393
 - Selektive Updates 153
 - Semantic Versioning 30–31, 259
 - Server Actions 717
 - Serverimplementierung 158
 - Serverkommunikation 71, 158, 195
 - Bibliotheken* 206
 - Fehlerbehandlung* 162
 - Server-Side Rendering 29, 289, 637
 - Serverzugriff 198
 - lesender* 198
 - schreibender* 199
 - Service-Worker 579
 - shadcn/ui 390
 - initialisieren* 391
 - installieren* 390
 - Komponenten* 391
 - Sicherheit 56
 - Sicherheitsmechanismus 71
 - Single-Page-Applikation 28, 393
 - Smart Component 167
 - SOAP 491
 - SPA → Single-Page-Applikation
 - Speedy Web Compiler 66
 - src-Verzeichnis 74
 - SSR → Server-Side Rendering
 - Standardport 159
 - startTransition 287
 - State 39–40, 47, 109, 441
 - Initialwert* 110
 - render* 109

State-Management	45, 441	TanStack Query	209, 213, 552
Änderungen	443	Daten modifizieren	215
Datenspeicher	442	Datensätze löschen	215
zentrales	469	Setup	210
State-Objekt	116	TanStack Router	393, 414
Statisches Asset	74	installieren	415
Storybook	436	konfigurieren	415
installieren	436	Routen definieren	417
Komponentenstory	437	Vite	415
Storys	437	TDD	
Streaming	761	Green	335
StrictMode	81–82, 152, 479	Red	335
Struktur	73	Refactor	335
aufräumen	80	TDD → Testgetriebene Entwicklung	
Dateien	73	Template-String	316
Komponenten	74	Terser	544
Style-Definitionen	74	Test	329
Verzeichnisse	74	automatisierter	330
Styled Components	316	Bibliothek	433
&&&	323	Dokumentation	329
Datei	319	Feedback	330
dynamisches Styling	322	Geschwindigkeit	330
Erweiterung	325	Grenzbereiche	343
Pseudoselektoren	321	Konfigurationsprobleme	330
styled-Funktion	324	Promise	345
Styleguides	85	Reihenfolge	330
style-Prop	311	Sicherheit	329
Stylesheet, globales	303	Umgebung	330
Styling	301	unabhängiger	330
submit-Event	226–227	Testgetriebene Entwicklung	335
Suspense	33, 213, 547	Throttling	483
Data Fetching	213	Thunk	471
Suspense for Data Fetching	34, 552	Timeout	148
SVG	379	Token	562
SWC → Speedy Web Compiler		Tokenbasierte Authentifizierung	562
Symbolischer Link	431	Touchoberfläche	33
SyncExternalStore-Hook	295	Transition-Hook	287
Syntax-Highlighting	303	Tree-Shaking	545
Synthetic Event	240	Triple A	336
Wrapper	240	Act	337
		Arrange	336
		Assert	337
T		try-catch	164
Tabelle, virtuelle	553	tsc	131
Tag-Funktion	316	Tutorial	27
Tagging-Funktion	319	Typecasting	102
Tailwind	325	TypeScript	50, 127, 129, 424
Installation	325	@types	134
Responsiveness	327	any	133
Standardklassen	327	Basisdatentypen	133
Theme	327	Basisfeatures	134
cachen	218	class	137
		Compiler	131

TypeScript (Forts.)	
<i>compilerOptions</i>	132
<i>constructor</i>	137
Dateiendung	130
Datentypen	135
Entwicklungsumgebung	130
<i>extends</i>	137
Features	133
Funktionen	136
Funktionskomponente	140
Generics	133
<i>implements</i>	137
Initialisierung	131
Interface	133, 139
Interface Merging	140
Klasse	137
Kommandozeile	131
Konfiguration	132
Konstruktor	137
Modulsystem	133
<i>private</i>	137
<i>protected</i>	137
<i>public</i>	137
React	129
<i>readonly</i>	137
State-Hook	142
<i>tsconfig.json</i>	132
Typdefinitionen	133
Type Alias	139
Variablen	134
Visual Studio Code	130
Vite	129
WebStorm	130
Zugriffsmodifikatoren	137
Typsicherheit	106
Typsystem	127
Dokumentation	127
Fehlersuche	128
Kategorien	128
Lesbarkeit	127
schwaches	127
Unterstützung	128
U	
Übersetzungen	515
Übersetzungsressource	518
UI State	47
UI-Bibliothek	369
Umgebungsvariablen	165
<i>.env</i>	165
Kommandozeile	165
VITE_	165
Uncontrolled Component	221
Fehlerbehandlung	228
Unit-Test	334
URL	
Änderung	395
useCallback	271
useContext	261
useDebugValue	282
useDeferredValue	283
useEffect	260
Abhängigkeiten	148, 153
<i>async/await</i>	164
Dependency	154
useEffectEvent	290
useId	289
useImperativeHandle	277
useInsertionEffect	295
useLayoutEffect	281
useMemo	537
useOptimistic	292
use-Präfix	296
useReducer	263
useRef	155, 222
useState	260
Rückgabewert	110
useSyncExternalStore	295
useTransition	287
in Bearbeitung	289
Utility-First-Framework	325
V	
V8	59
Validierung	228
<i>clientseitige</i>	228
<i>serverseitige</i>	228
var	135
Vercel	638
Verdaccio	433
View-Layer	27
Virtual DOM	42
Virtuelle Tabelle	553
Virtuelles Scrollen	554
Visual Studio Code	59
Vite	28, 60, 541
<i>shadcn/ui</i>	390
<i>TanStack Router</i>	415
Vite PWA	578, 590
Vite Scripts	68
<i>build</i>	69
<i>preview</i>	69

Vitest 45, 329, 424

- .not* 339
- afterAll* 341
- afterEach* 341
- asynchrone Operationen* 345
- beforeAll* 341
- beforeEach* 340
- describe* 339
- Ende-zu-Ende-Tests* 366
- Exceptions* 343
- expect* 337
- externe Abhängigkeit* 361
- Formulare testen* 358
- Installation* 332
- Interaktion testen* 357
- it* 336
- Kommandos* 333, 352
- Komponenten* 353
- Matcher* 337–338
- Mock-Backend* 361
- Mocks* 347
- only* 342
- Organisation* 334
- Props* 354
- Referenz-Snapshot* 351
- rejects* 346
- resolves* 346
- Serverabhängigkeit* 361
- Serverfehler simulieren* 364
- Serverkommunikation* 347
- Setup-Routinen* 340
- skip* 341
- Snapshot* 46, 348, 352
- Spy-Funktion* 348, 358
- Tear-down-Routinen* 340
- test* 336
- Tests gruppieren* 339
- Testsuites* 339
- toThrow* 343
- Typüberprüfung* 344
- überspringen* 341
- Zufallszahlen* 347

Vitest Browser Mode 366

Vue.js 29, 54

W

Walke, Jordan 30

Webdriverio 366

Webpack 608

- Dev-Server* 84

Webserver 56

WebSockets 154

WebStorm 59

Wiederverwendung 423

Workbox 590

Wurzelement 94

Wurzelknoten 34

Wurzelkomponente 80, 83

X

XHP 30

XSS-Injections 515

XSS-Protection 97

Y

Yarn 63

- Caching* 64
- Integrität* 64
- Sicherheit* 64

Z

Zentrales State-Management 295, 469

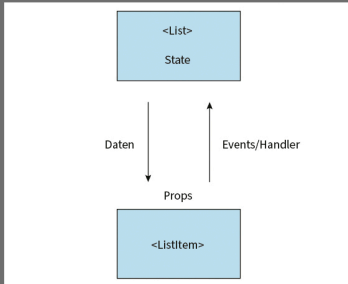
Zod 207, 254, 358, 386

- infer* 728
- safeParse* 729

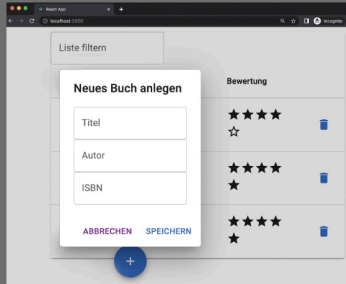
Zustand 39

So geht moderne Frontend-Entwicklung mit React

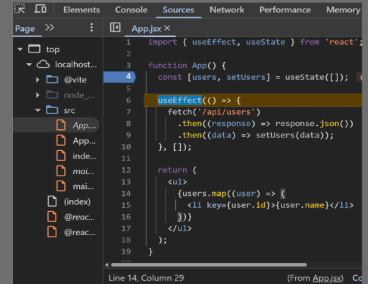
Skalierbar, flexibel, performant: React ist die führende JavaScript-Bibliothek für moderne Benutzeroberflächen. Dieses Buch begleitet Sie durch alle Phasen der Entwicklung – von den Grundlagen bis zur professionellen Anwendung. Anhand vieler Beispiele lernen Sie den modernen Standard-Stack rund um Next.js,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

