2 Schnelldurchgang – React im Überblick

Bevor wir in die Details von React gehen, möchten wir dir in diesem Kapitel die wichtigsten Features von React im Schnelldurchgang an einem sehr einfachen Beispiel zeigen. Das Beispiel kannst du auf der Plattform CodeSandbox, die Online-Editoren für JavaScript anbietet, nachvollziehen, ohne dass du dafür etwas bei dir lokal auf deinem Computer installieren musst. (CodeSandbox selbst ist übrigens auch mit React gebaut.)

Vorbereitung

Um das Beispiel zu entwickeln, kannst du in CodeSandbox ein neues React-Projekt anlegen. Dazu öffnest du *https://codesandbox.io/*, klickst auf »Create Sandbox« und wählst dann unter »Popular Templates« »React« aus. Daraufhin wird ein Projekt angelegt, in dem die Abhängigkeiten auf die React-npm-Module bereits hinzugefügt sind und einige weitere Dateien angelegt sind.

Je nach Einstellung findest du auf der linken Seite einen Explorer mit allen Dateien des Projekts, in der Mitte einen Editor (der sich wie VS Code verwenden lässt!) und rechts die Darstellung deiner Anwendung in einem simulierten Browser. Sobald du den Code deiner Anwendung veränderst, wird die Anwendung übersetzt (z.B. JSX nach ES5) und die Darstellung im simulierten Browser automatisch aktualisiert.

Für diesen Schnelleinstieg wollen wir eine ganz einfache Hello-World-Anwendung bauen, die nur aus einer einzigen Komponente besteht. Komponenten sind das zentrale Element in React und wie wir später noch sehen werden, sind Anwendungen in React nichts weiter als eine Menge von Komponenten.

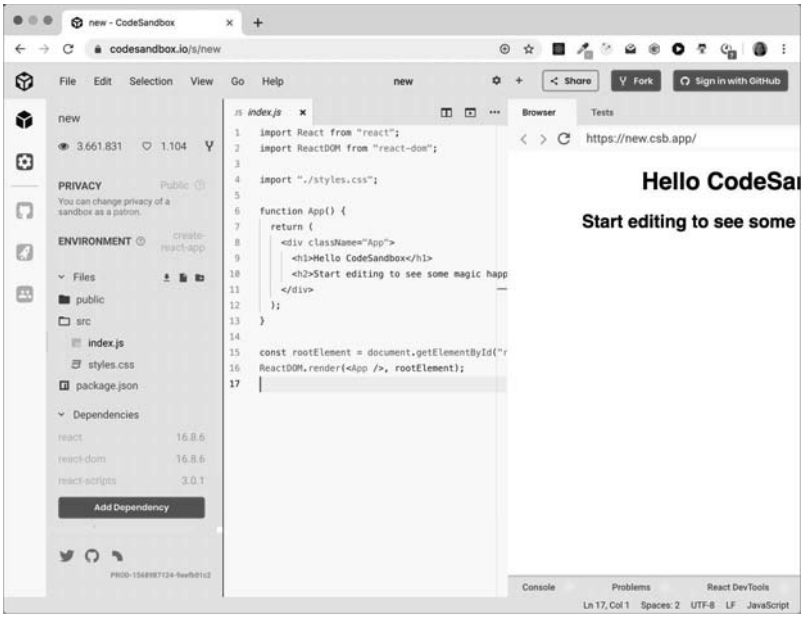

Abb. 2–1
Der Online-Editor CodeSandbox mit Datei-Explorer, Editor und Vorschau der Anwendung

Die Komponente unserer Anwendung, `Greeter`, besteht aus einem Eingabefeld, in dem ein Gruß eingegeben werden kann. Der eingegebene Gruß darf eine bestimmte Länge nicht überschreiten. Die Anzahl der verbleibenden Zeichen steht unterhalb des Eingabefeldes. Außerdem gibt es einen Knopf, der den aktuellen Gruß ausgibt. Aktiv ist der Knopf allerdings nur, wenn der Gruß »gültig« ist, das bedeutet, er enthält mindestens ein Zeichen und die Länge ist kleiner oder gleich der erlaubten Maximallänge.

Beispiel: die Greeter-Komponente

Der erste Schritt unserer Anwendung sieht wie folgt aus, du kannst diesen Code so, wie er da steht, in die `index.js`-Datei deiner Sandbox einsetzen:

```
import React from "react";
export default function Greeter(props) {
  const [greeting, setGreeting] = React.useState("");

  function handleGreetClick() {
    alert(`Hello, ${greeting}`);
  }

  const charsRemaining = props.maxLength - greeting.length;
  const greetingInvalid = greeting.length === 0
                                || charsRemaining < 0;

  return (
    <div>
      Greeting:
```

```
        <input value={greeting}
          onChange={e => setGreeting(e.target.value)} />
        <span>{charsRemaining}</span>

        <button disabled={greetingInvalid}
                  onClick={handleGreetClick}>Greet
        </button>
      </div>
    );
  }
```

Sehen wir uns den Code der Komponente Schritt für Schritt an. Eine Komponente in React ist eine JavaScript-Funktion. In dieser Komponentenfunktion ist alles enthalten, was die Komponente benötigt, um sich darstellen zu können: sowohl die Logik als auch die UI.

Beschreibung der UI mit JSX

Die UI, die unsere Komponente darstellen soll, findest du im `return`-Statement der Komponente. Dort haben wir mit einer HTML-artigen Syntax die Elemente beschrieben, die zur Laufzeit im Browser dargestellt werden sollen. Dieser HTML-artige Code, JSX genannt, wird zur Build-Zeit von einem Compiler (Babel oder TypeScript) in gültiges JavaScript übersetzt. Für uns hat dieser Ansatz den Vorteil, dass wir keine Template-Sprache lernen und verwenden müssen (auch wenn dieses Vorgehen am Anfang sicherlich gewöhnungsbedürftig und vielleicht sogar abschreckend sein mag).

Variablen verwenden

Innerhalb des JSX-Codes können wir auf JavaScript-Variablen und -Funktionen zugreifen. Im Button, der den Gruß anzeigen soll, geben wir damit an, ob der Button aktiv oder inaktiv sein soll. Außerdem geben wir eine Callback-Funktion an, die ausgeführt werden soll, sobald auf den Button geklickt wird. Diese Eigenschaften eines Elementes werden in React Properties genannt. Sowohl `disabled` als auch `onClick` sind somit Properties des Button-Elements, genauso wie `value` und `onChange` Properties des `input`-Felds sind.

Properties

Auch unsere Komponente selbst nimmt ein Property entgegen, mit dem der Verwender der Komponente angeben kann, wie lang der Gruß maximal sein kann (`maxLength`). Properties werden einer Komponente über das erste Funktionsargument, `props`, in einem Objekt übergeben, sodass wir in unserem Beispiel auf das einzige Property der Komponente mit dem Ausdruck `props.maxLength` zugreifen können und die Anzahl verbleibender Zeichen sowie das Enablement des Buttons berechnen können.

Zustand einer Komponente

An dem `input`-Element geben wir mit dem `onChange`-Property eine Callback-Funktion an, die aufgerufen wird, wenn sich das `input`-Feld zur Laufzeit ändert, also beispielsweise, weil ein Zeichen eingegeben wurde. Da React über keine Möglichkeit verfügt, Daten aus der UI automatisch an ein Model zu binden (2-Wege-Databinding), müssen wir uns selber darum kümmern.

State

Das Model einer Komponente wird in React Zustand oder State genannt und wird mit der Funktion `React.useState` erzeugt, der wir als Parameter einen initialen Wert für den Zustand übergeben (in unserem Beispiel einen Leerstring). Diese Funktion liefert uns ein Array mit zwei Einträgen zurück: den aktuellen Wert des Models (in unserem Fall der String mit dem Gruß) und eine *Setter-Funktion* zum Ändern des Models.

Ändern des Zustands löst Rendern aus

Das Besondere an dem Zustand ist: Sobald wir diesen mit der zurückgelieferten Setter-Funktion ändern, wird die gesamte Komponentenfunktion von React erneut aufgerufen und ausgeführt, sodass sie ihre aktualisierte UI zurückliefern kann.

Da wir die Setter-Funktion als Reaktion auf Änderungen im Eingabefeld aufrufen, wird die Komponentenfunktion folglich nach jeder Veränderung im Eingabefeld neu aufgerufen. Bei diesen folgenden Aufrufen der `Greeter`-Funktion liefert `useState` nun jeweils den aktualisierten Wert des Models zurück (und nicht mehr den initialen Wert), den React für uns im Hintergrund verwaltet und aufbewahrt.

Innerhalb unserer Funktion können wir abhängig vom Zustand nun eine komplett neue UI beschreiben. Dabei übernehmen wir zum Beispiel den Wert (aktueller Gruß) in das `input`-Feld (`value`-Property) und setzen das Enablement des Buttons neu. Auch die Anzeige der verbleibenden Zeichen für den Gruß wird aktualisiert.

Deklarative UI

Eine Stärke dieses Vorgehens, der deklarativen Beschreibung der UI und des erneuten Ausführens der kompletten Funktion nach einer einzelnen Änderung, ist es, dass wir nicht in inkonsistente Zustände in der UI kommen können. Alle UI-Elemente, die den Zustand oder Teile davon oder davon abgeleitete Werte (Anzahl verbleibender Zeichen) anzeigen, werden in jedem Fall vollständig aktualisiert.

Nach dem Ausführen der Funktion aktualisiert React anhand der zurückgelieferten UI die Darstellung im Browser. Und obwohl wir immer die komplette UI der Komponente an React zurückgeben, nimmt React nur so wenige Änderungen im DOM wie nötig vor, damit dort die Änderungen so schnell und so effizient wie möglich erfolgen.

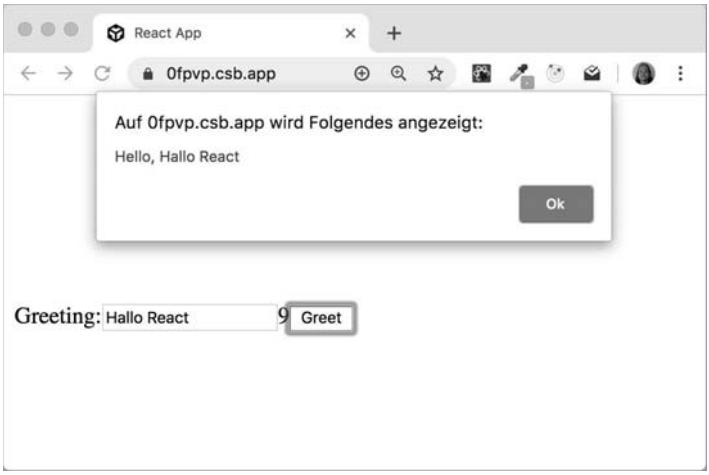

Abb. 2–2
Die Hello-World-Anwendung

Rendern der Anwendung im Browser

ReactDOM.render

Beim Starten unserer Anwendung müssen wir React nun noch angeben, dass unsere Komponente überhaupt dargestellt werden soll. Dazu verwenden wir die Funktion `ReactDOM.render`. Dieser Funktion übergeben wir unsere `Greeter`-Komponente, ebenfalls in der JSX-Notation, wie oben schon bei `button`, `input` etc. gesehen. Dabei geben wir auch das Property `maxLength` an. Außerdem erwartet die Funktion als zweiten Parameter einen DOM-Knoten aus der aktuellen Seite, unterhalb dessen die Komponente dargestellt werden soll. In unserem Beispiel ist dieser DOM-Knoten ein `div`-Element mit der Id `root`, den du in der erzeugten index.html-Datei im `public`-Ordner der Sandbox findest.

Rendern

Zum Rendern der Komponente füge nun bitte den Import für `ReactDOM` und den entsprechenden `render`-Aufruf in der `index.js`-Datei hinzu:

```
import React from "react";
import ReactDOM from "react-dom";

export default function Greeter(props) {
  ...
}

ReactDOM.render(<Greeter maxLength={20} />,
  document.getElementById("root")
);
```

Nun wird die Komponente im Browser von CodeSandbox angezeigt und du kannst sie direkt dort ausprobieren. Eine Sandbox mit der fertigen Beispielanwendung findest du unter *https://codesandbox.io/s/react-buch-schnell-01-0fpvp*.

2.1 Zusammengefasst: Unterschiede zwischen Hooks- und Klassen-API

In diesem Buch verwenden wir fast ausschließlich die Hooks-API, die erstmals Ende 2018 vorgestellt und im Februar 2019 mit React 16.8 in einer stabilen Version veröffentlicht wurde.

Hooks sind Funktionen, mit denen du nahezu alle React-Features in Funktionskomponenten verwenden kannst, die zuvor der Klassen-API vorbehalten waren (zum Beispiel Arbeiten mit State). Falls du schon die Klassen-API, zum Beispiel aus der ersten Auflage dieses Buchs, kennst, geben wir dir hier zur groben Orientierung eine kurze Gegenüberstellung zwischen Klassen- und Hooks-API:

- Zustand/State einer Komponente: Dafür ist der `useState`-Hook (siehe Kapitel 4) zuständig. Das State-Objekt gibt es in Funktionskomponenten nicht, für einen komplexeren Zustand kannst du den `useReducer`-Hook verwenden (siehe Kapitel 6).
- Die Lifecycle-Methoden aus den Klassenkomponenten haben kein direktes Pendant in der Hooks-API. Stattdessen verwendest du den `useEffect`-Hook (siehe Kapitel 7).
- Die Funktionalität von Pure Components sowie der `shouldComponentUpdate`-Methode können durch den `useMemo`-Hook umgesetzt werden (siehe Abschnitt 5.4).
- Als Ersatz für die Verwendung der Provider-Komponente, um auf React Context zuzugreifen, gibt es den `useContext`-Hook (siehe Abschnitt 9.6).
- Higher-Order Components (HOCs) gibt es weiterhin, allerdings gibt es mit der Möglichkeit, eigene Hooks zu bauen, eine API, die möglicherweise einfacher und besser zu verstehen ist. Möglicherweise werden HOCs deswegen künftig seltener eingesetzt werden.
- Die Aufgaben der `connect`-Funktion von Redux können durch die Hooks `useSelector` und `useDispatch` übernommen werden (siehe Kapitel 10).
- Die `withRouter`-HOC aus dem React-Router-Projekt sowie die `render`- und `component`-Properties der `Route`-Komponente werden durch diverse neue Hooks ersetzt (siehe Kapitel 9).

Um die Auswirkungen auf den Code einer React-Anwendung zu sehen, haben wir die fertige Beispielanwendung im Repository auch in einer Variante mit der Klassen-API abgelegt. So kannst du React-Code mit Klassen- und Hooks-API vergleichen.

2.2 Ein React-Projekt beginnen mit Create React App

Um ein ernsthaftes React-Projekt zu beginnen und zu entwickeln ist eine ganze Reihe an Infrastrukturkomponenten erforderlich:

- Neuerer JavaScript-Code, der (noch) nicht von allen Browsern verstanden wird, muss in eine ältere JavaScript-Sprachversion zurückübersetzt werden, die von allen Browsern, auf denen deine Anwendung laufen soll, unterstützt wird. Das ist in der Regel ES5. Zudem muss der React-proprietäre JSX-Code (siehe Kapitel 4) in gültiges und verständliches JavaScript übersetzt werden. Zum Kompilieren des Sourcecodes wird in der Regel Babel oder TypeScript verwendet.
- Auch für die Arbeit mit JavaScript-Modulen wird ein Tooling benötigt. Zum einen verstehen nicht alle Browser das native JavaScript-Modulsystem, das 2015 eingeführt wurde. Zum anderen gibt es weitere Modulsysteme, in denen Module vorliegen können, die du möglicherweise in deiner Anwendung verwendet möchtest. Ein populäres Modulsystem ist das aus NodeJS stammende RequireJS, für das es eine Reihe von Modulen gibt, die auch im Browser laufen und die du bei Bedarf in deiner Anwendung verwenden kannst. Um deine eigenen und die von dir verwendeten Module für den Browser aufzubereiten, wird ein Bundler benötigt. Während ein Compiler sich um das Übersetzen eines Moduls von einer JavaScript-Sprachversion in eine ältere kümmert, sorgt der Bundler dafür, dass deine Module, unabhängig davon welches Modulsystem du verwendest, auch im Browser funktionieren. Eines der prominentesten Bundler ist Webpack.
- Für das Testen deiner Anwendung benötigst du ein Testbibliothek, das in der Lage ist, React-Komponenten zu testen. Die Tests sollen natürlich regelmäßig – zum Beispiel im CI-Build – ausgeführt werden.
- Während du deine Anwendung entwickelst, benötigst du einen Webserver, über den du deine Anwendung testweise betreiben kannst. Darüber sollten Änderungen, die du vornimmst, auch sofort im Browser sichtbar werden, ohne dass du viel Mühe damit hättest. Du benötigst also Werkzeuge für die Automatisierung. Hier kommt der Webpack Development Server ins Spiel, der dafür sorgt, dass bei Änderungen an deinem Sourcecode die Anwendung neu kompiliert und im Browser aktualisiert wird.
- Neben der Konfiguration dieser Tools für den Entwicklungsprozess benötigt man in der Regel auch eine spezialisierte Konfiguration für das Bauen einer Anwendung, die dann auch in Produktion ausgeliefert wird. Während bei der Entwicklung eher die Build-

Dauer und der Entwicklungskomfort, etwa zum Debuggen, im Vordergrund stehen, ist beim Bauen der Anwendung für die Produktion eher entscheidend, dass die fertigen Artefakte zum Beispiel möglichst klein werden.

Das Einrichten eines Projekts mit all den benötigten Tools sowie das manuelle Pflegen ihrer unterschiedlichen Versionen und Konfigurationen kann sehr zeitaufwendig sein. Aus diesem Grund gibt es das Projekt Create React App[1], das dir diese Aufgaben abnimmt. Mit Create React App kannst du ein neues React-Projekt erzeugen lassen. Das Projekt enthält dann fertige und sehr ausgereifte Konfigurationen für Babel (auf Wunsch TypeScript) und Webpack. Auch Unterstützung für CSS ist enthalten und es werden für alle Artefakte SourceMaps erzeugt, sodass du trotz proprietärem JSX-Code weiterhin zum Beispiel im Browser debuggen kannst (siehe Abb. 2–3). Darüber hinaus enthält das generierte Projekt Konfigurationen für einen Produktionsbuild, der nach den jeweils gültigen Best Practices implementiert ist und sehr gute Ergebnisse erzielt.

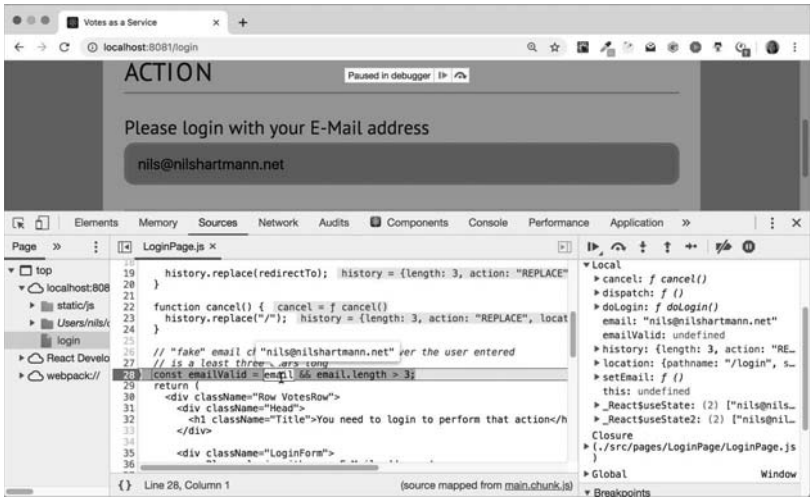

Abb. 2–3
Debuggen einer React-Komponente in Chrome

Um ein neues Projekt zu starten, kannst du mit dem Tool npx (Bestandteil von npm) create-react-app ausführen. Dazu öffnest du eine Kommandozeile und gehst in das Verzeichnis, in dem das neue Projekt (in einem neuen Verzeichnis) angelegt werden soll:

```
npx create-react-app PROJEKT_NAME
```

1 *https://facebook.github.io/create-react-app/*

Mit diesem Aufruf erstellt Create React App ein neues Verzeichnis und legt darin die `package.json`-Datei mit den notwendigen Abhängigkeiten an. Außerdem werden schon einige beispielhafte Source-Dateien angelegt, sodass du gleich mit dem Entwickeln anfangen kannst.

ESLint

Bestandteil des erzeugten Projekts ist auch die Konfiguration von ESLint[2], einem populären Linter für JavaScript. Ein Linter ist ein Tool zur statischen Codeanalyse, das Probleme im Sourcecode aufdecken kann (z.B. Verwendung nicht deklarierter Variablen und ungenutzte Variablen). ESLint lässt sich mit Regeln beliebig konfigurieren und es gibt auch fertige Regeln für die Entwicklung von React-Anwendungen. Einige Regeln sind in dem mit Create React App erzeugten Projekt bereits eingetragen, sodass du bei typischen Codeproblemen Hinweise darauf beim Build bekommst. Viele IDEs und Editoren (z.B. VS Code oder Webstorm) zeigen dir die Hinweise auch direkt beim Bearbeiten einer Datei an.

Abb. 2–4
Darstellung von Problemen aus ESLint in Visual Studio Code

Auch unsere Beispielanwendung haben wir initial mit Create React App erzeugt und nur die generierten Source-Artefakte gelöscht bzw. für unsere Zwecke angepasst.

2 *https://eslint.org/*

2.2.1 Progressive Web Apps mit React

Wenn du ein Projekt mit Create React App erstellst, ist der dabei erzeugte Code bereits für die Entwicklung einer Progressive Web App[3] (PWA) ausgelegt. Progressive Web Apps sind Webanwendungen, die über Features verfügen, die in der Vergangenheit nativen Anwendungen vorbehalten waren. Dazu gehört zum Beispiel die Möglichkeit, die Anwendung auf dem Homescreen eines Mobilgerätes installieren zu können. Dazu muss die Anwendung eine Manifestdatei enthalten, die zum Beispiel darüber Auskunft gibt, welchen Titel die Anwendung anzeigen und welches Icon sie verwenden soll. PWAs sind außerdem auch offline-fähig, sodass sie auch geöffnet und verwendet werden können, wenn keine Netzverbindung besteht. Dazu werden *Service Worker*[4] verwendet, die Daten cachen können.

Beide Artefakte, das Manifest und ein Service Worker, werden von Create React App automatisch erzeugt. Der generierte Service Worker ist aber per Default nicht aktiviert, da Entwicklung und Betrieb von Anwendungen mit einem Service Worker nicht unproblematisch ist. Du kannst den Service Worker aber in der generierten `index.js`-Datei jederzeit einschalten.

Damit hast du eine gute Basis, um deine React-Anwendung bei Bedarf als Progressive Web App zu bauen.

2.2.2 Hintergrund: react-scripts

Eine Besonderheit von Create React App bzw. den Projekten, die mit Create React App erzeugt wurden, ist, dass es die Abhängigkeiten auf die Tools (Babel, Webpack etc.) nicht direkt in deine `package.json`-Datei einträgt. Stattdessen wird ein Verweis auf ein Modul mit dem Namen `react-scripts` eingetragen. Dieses Modul wiederum enthält dann sämtliche Konfigurationen und bestimmt, welche Tools in welcher Version zum Einsatz kommen. Das Modul wird von dem Team gepflegt, das Create React App entwickelt. Gibt es neue Versionen der abhängigen Tools, zum Beispiel eine neue Webpack-Version, oder Verbesserungen in der Konfiguration, zum Beispiel eine Optimierung für den Build-Prozess, wird `react-scripts` von den Entwicklern angepasst und eine neue Version veröffentlicht. Du kannst dann von den Neuerungen auch in deinem bestehenden Projekt profitieren, indem du dort einfach die Version von `react-scripts` in deiner `package.json`-Datei anpasst.

---

3 *https://developers.google.com/web/progressive-web-apps/*
4 *https://developer.mozilla.org/de/docs/Web/API/Service_Worker_API/*

Anpassen der Konfiguration

Die Kehrseite der Medaille ist, dass du von der Konfiguration weitgehend abgeschnitten bist. Bis auf wenige Parameter kannst du die Konfiguration nicht beeinflussen. Zwar ist die Konfiguration sehr gut und für viele Einsatzzwecke auch völlig ausreichend, aber natürlich kann es immer vorkommen, dass du doch an der Konfiguration arbeiten musst. Für diesen Fall gibt es zwei Möglichkeiten: Zum einen kannst du das Projekt `react-scripts` auf GitHub klonen, anpassen und eine eigene Variante für dein Projekt davon erzeugen, die du dann in die `package.json`-Datei einträgst. Sofern du später feststellst, dass – z.B. durch eine neuere Version – `react-scripts` doch wieder für dich ausreichend ist, kannst du wieder auf `react-scripts` umstellen.

»Auswerfen« von react-scripts

Falls dieser Ansatz für dich nicht ausreichend ist, kannst du das Projekt `react-scripts` »auswerfen« lassen. Dieses Verfahren heißt in Create React App »eject«. Damit werden alle Konfigurationen und Abhängigkeitsbeschreibungen direkt in dein Projektverzeichnis kopiert und die Abhängigkeit auf `react-scripts` entfernt. Dein Projekt sieht dann so aus, als ob dort direkt (wie in einem »traditionellen« Projekt) die Konfigurationen und Abhängigkeiten eingetragen worden wären, sodass du dort beliebig Anpassungen machen kannst. Der Nachteil dieses Ansatzes ist, wenn du einmal »ausgeworfen« hast, kommst du nicht mehr zurück. Du bist dann von den Neuerungen in `react-scripts` abgeschnitten.

2.2.3 React-Anwendungen in Produktion betreiben

React-Anwendungen bestehen aus HTML-, JavaScript- und CSS-Dateien. Dazu kommen unter Umständen weitere statische Assets wie Bilder oder Fonts.

Wenn du dein Projekt mit Create React App erzeugt hast, kannst du mit dem `npm`-Script `build` dir eine fertige Anwendung bauen lassen, die für den produktiven Einsatz optimiert ist, unter anderem sind JavaScript- und CSS-Code minifiziert und unter Umständen auch auf mehrere Dateien aufgeteilt:

```
npm run build
```

Die erzeugen Dateien werden in das `build`-Verzeichnis geschrieben. Zum Veröffentlichen deiner Anwendung über einen Webserver musst du dieses Verzeichnis auf deinen Webserver kopieren und ihn als Web-Root für den gewünschten Server, die Domain oder den Pfad festlegen.

2.3 Zusammenfassung

In diesem Kapitel haben wir uns anhand einer sehr einfachen Beispielanwendung die grundlegenden Konzepte von React im Code angesehen.

- Komponenten in React werden als Funktionen geschrieben, die von außen Parameter (Properties) bekommen und intern einen Zustand halten. Die Komponente ist verantwortlich, dafür eine UI zu erzeugen.
- Mit dem Online-Editor CodeSandbox kannst du ohne etwas installieren zu müssen, React ausprobieren. Den Editor kannst du auch gut benutzen, um mit anderen über Code zu diskutieren (zum Beispiel für Bug-Reports).
- »Echte« React-Projekte kannst du mit Create React App automatisiert anlegen. Projekte, die damit begonnen werden, verfügen über eine vollständige Konfiguration für den Build-Prozess sowie die Verwaltung der Abhängigkeiten.

gistrieren neuer Abstimmungen (PUT). In beiden Fällen wollen wir aus der Antwort vom Server noch ein JSON-Objekt erzeugen, das wir per Callback-Funktion an den Initiator des Server Requests zurückgeben.

Um diesen (technischen) Code zentral vorzuhalten, erstellen wir sowohl für Lese- als auch für Schreibzugriffe zwei Hilfsfunktionen. Dafür lege bitte die neue Datei `src/backend.js` an, in der wir die Funktionen zur Serverkommunikation implementieren. Du kannst dort jetzt schon die URL zu unserem API-Server als Konstante hinterlegen:

```
const BACKEND_URL = "http://localhost:3000";
```

Nun implementieren wir die Hilfsfunktionen zum Lesen von Daten als HTTP-*GET*-Aufruf. Die Funktion soll `fetchJson` heißen und erwartet einen Parameter, nämlich den aufzurufenden Pfad (also den Teil der URL, der hinter dem Servernamen kommt). Als zweiten Parameter können (optional) weitere Eigenschaften für den zugrunde liegenden fetch-Aufruf übergeben werden. Die Funktion wird als async-Funktion implementiert und liefert ein Promise zurück, das aufgelöst wird, sobald die Antwort vom Server gekommen ist und in ein JSON-Objekt umgewandelt wurde. Damit die Funktion außerhalb der `backend.js`-Datei sichtbar ist, muss diese exportiert werden. Füge jetzt also bitte folgenden Code in die `backend.js`-Datei ein:

Die Funktion fetchJson (backend.js)

```
export async function fetchJson(path, options) {
  const url = `${BACKEND_URL}${path}`;

  const response = await fetch(url, options);
  if (!response.ok) {
    throw new Error(`Response not OK: ${response.status}`);
  }
  return await response.json();
}
```

Beim Ausführen der Funktion wird der HTTP-Aufruf an den übergebenen Pfad auf unserem API-Server durchgeführt. Das vom Server zurückgeschickte Ergebnis wird in ein JSON-Objekt umgewandelt. Falls der HTTP-Status der Antwort nicht Status OK (200) entspricht, werfen wir einen Fehler, sodass der Aufrufer der Funktion darauf reagieren kann.

Daten auf den Server schreiben

Implementieren wir nun die Funktion für die schreibenden Serverzugriffe, die wir `sendJson` nennen. Analog zur `fetchJson`-Funktion erwartet die Funktion als ersten Parameter den Pfad. Zusätzlich muss der Aufrufer die HTTP-Methode (beispielsweise *POST* oder *PUT*) übergeben und das Objekt, das als Body im JSON-Format an den Server geschickt werden soll. Die Funktion ist ebenfalls als asynchrone Funktion implementiert und liefert ein Promise zurück, das aufgelöst

wird, sobald eine Antwort vom Server zurückgekommen ist und in ein JSON-Objekt umgewandelt wurde. Auch diese Funktion wird exportiert, um sie außerhalb der `backend.js`-Datei sichtbar zu machen. Die vollständige Funktion sieht dann so aus:

Die Funktion sendJson (backend.js)

```
export async function sendJson(method, path, payload = {}) {
  const url = `${BACKEND_URL}${path}`;

  const response = await fetch(url, {
    method: method,
    body: JSON.stringify(payload),
    headers: {
      Accept: "application/json",
      "Content-Type": "application/json"
    }
  });

  if (!response.ok) {
    throw new Error(`Response not OK: ${response.status}`);
  }

  return await response.json();
}
```

Die Funktion konvertiert das mit dem Parameter `payload` übergebene Objekt in einen JSON-String und setzt sie als Body für die Serveranfrage. Außerdem werden für den Serverzugriff zwei HTTP-Header gesetzt: `Accept` und `Content-Type`.

Schritt 2: Votes vom Server abfragen

Um die Kommunikation mit dem Server durchzuführen, legen wir nun die neue Komponente `VoteListPage` an. Dazu gehen wir in mehreren Schritten vor. Sehen wir uns als Erstes an, was nötig ist, um die `vote`-Objekte von unserem Server zu lesen. Dazu schreiben wir zunächst eine Funktion, die die Daten lädt und dann in den Zustand der Komponente setzt. Demnach wird die Komponente neu gerendert, sobald die Daten vorhanden sind, und die Komponente kann die Daten – mithilfe von `VoteController` – anzeigen. Solange noch keine Daten vorhanden sind, wird dem Benutzer eine entsprechende Meldung angezeigt (`VoteLoadingIndicator`).

Die initiale Version von unserer `VoteListPage` sieht wie folgt aus:

Die Komponenten VoteLoadingIndicator und VoteListPage

```
import React from "react";
import { fetchJson, sendJson } from "../backend";

function VoteLoadingIndicator() {
  return (
    <div className="Row VotingRow Spacer">
```

```
      <h1 className="Title">Votes are loading...</h1>
    </div>
  );
}

export default function VoteListPage() {
  const [allVotes, setAllVotes] = React.useState(null);

  async function loadVotes() {
    const votes = await fetchJson("/api/votes");
    setAllVotes(votes);
  }


  if (!allVotes) {
    return <VoteLoadingIndicator />;
  }

  return (
    <VoteController
      allVotes={allVotes} ... />
  );
}
```

Die Komponente wird beim initialen Rendern also den `VoteLoadingIndicator` anzeigen, da noch keine Daten geladen wurden; dazu müsste die `loadVotes`-Funktion aufgerufen werden. Da die Daten automatisch geladen werden sollen, sobald die Komponente das erste Mal verwendet wird, wäre ein naiver Ansatz, die `loadVotes`-Funktion direkt vor dem `return` aufzurufen. Dann würden darin die Daten geladen und sobald sie vom Server kommen, würden sie in den Zustand gesetzt und die Komponente erneut gerendert werden. *Das ist in React allerdings verboten! Funktionskomponenten müssen seiteneffektfrei sein.* Zu Seiteneffekten gehören beispielsweise das Manipulieren des DOM, das Öffnen (oder Schließen) einer Websocket-Verbindung oder eben Serveraufrufe.

Seiteneffekte mit useEffect

Aus diesem Grund gibt es den Hook `useEffect`, mit dem wir uns in den Lebenszyklus der Komponente einhängen können. Mit diesem Hook können wir eine Callback-Funktion angeben, die von React aufgerufen wird, sobald das Rendering der Komponente abgeschlossen ist. Zu diesem Zeitpunkt, *nach* dem Rendering, sind Seiteneffekte erlaubt, sodass wir innerhalb der Callback-Funktion unseren Server-Call durchführen können. Wichtig dabei ist, dass die übergebene Callback-Funktion keine asynchrone Funktion sein darf. Auch aus diesem Grund haben wir das Laden der Votes in eine eigene Funktion ausgelagert (der andere Grund ist, dass wir diese Funktion später noch verwenden werden, um die Daten zu aktualisieren).

Die Funktion zum Laden der Votes (`loadVotes`) wartet, bis die Daten vom Server gekommen sind, und setzt diese mittels `setVotes` in den Zustand, woraufhin sich die Komponente neu rendert.

Mit dem `useEffect`-Hook könnte die Komponente nun wie folgt aussehen (führt allerdings so zu einer Endlosschleife, die wir gleich auflösen):

Fehlerhafte Verwendung von useEffect

```
function VoteListPage() {
  const [allVotes, setAllVotes] = React.useState(null);

  async function loadVotes() {
    const votes = await fetchJson("/api/votes");
    setAllVotes(votes);
  }

  React.useEffect(() => {
    loadVotes();
  });

  if (!allVotes) {
    return <VoteLoadingIndicator />;
  }

  return <VoteController ... />;
}
```

Der Lebenszyklus der Komponente sieht jetzt wie folgt aus:

1. Die Funktion `VoteListPage` wird von React erstmalig aufgerufen (Render-Phase). Dabei wird der `allVotes`-State mit `null` vorbelegt und mit `useEffect` wird die `loadVotes`-Callback-Funktion als Effekt, der nach dem Rendern ausgeführt werden soll, registriert. Da noch keine Daten geladen wurden, wird die `VoteLoadingIndicator`-Komponente zurückgeliefert.
2. React schließt das Rendern ab, schreibt also die benötigten Änderungen in den nativen DOM und führt die mit `useEffect` hinterlegte Callback-Funktion aus (Commit Phase). In unserem Beispiel beginnt also das Laden der Daten in `loadVotes`.
3. Der Server antwortet auf den Request und `loadVotes` setzt die geladenen Umfragen in den State.
4. Die Funktion `VoteListPage` wird erneut von React aufgerufen, da sich der State verändert hat (Render-Phase). Der State enthält nun die geladenen Umfragen und die Funktion gibt die `VoteController`-Komponente zurück.
5. React schließt das Rendern ab und aktualisiert den DOM entsprechend (Commit-Phase).

6. Achtung: Der `useEffect`-Hook wird erneut ausgeführt! Die Daten werden erneut geladen, der Zustand neu gesetzt etc. Wir befinden uns also in einer Endlosschleife. Dieses Problem lösen wir gleich.

Schritt 3: Laden der Daten abschließen

Abhängigkeiten für useEffect angeben

Wie in der Zusammenfassung gesehen, funktioniert unsere Komponente nur scheinbar, da wir uns in einer Endlosschleife befinden: die Komponente wird gerendert, Hook wird ausgeführt, Daten werden geladen und in den Zustand gesetzt, Komponente wird erneut gerendert, Hook wird ausgeführt, Daten werden geladen und in den Zustand gesetzt ... Wir müssen React also noch angeben, unter welchen Bedingungen unsere Callback-Funktion ausgeführt werden soll. Das machen wir mit dem zweiten Parameter von `useEffect`. Darin geben wir in einem Array Werte als *Abhängigkeiten* an. Nur wenn sich eine der Abhängigkeiten, also einer der Werte, zwischen zwei Renderzyklen verändert, wird der Hook erneut ausgeführt. In unserem Beispiel möchten wir, dass der Hook in jedem Fall nur ein einziges Mal nach dem ersten Rendern ausgeführt wird, deswegen geben wir ein leeres Array ein (ein Beispiel für die Verwendung von Werten in dem Array findest du weiter unten, in Abschnitt 7.2):

Beispiel: useEffect mit Abhängigkeiten

```
function VoteListPage() {
  const [allVotes, setAllVotes] = React.useState(null);

  async function loadVotes() {
    const votes = await fetchJson("/api/votes");
    setAllVotes(votes);
  }

  React.useEffect(
    () => { loadVotes() },
    [] // nur einmaliges Rendern erzwingen
  );

  if (!allVotes) { return ... }

  return <VoteController votes={allVotes} />;
}
```

Um die neue Komponente `VoteListPage` auch zu verwenden, ersetzen wir in der `index.js` den `VoteController` durch die `VoteListPage`-Komponente. Auch die hartcodierten Beispieldaten können wir dort nun löschen:

```
import VoteListPage from "./components/VoteListPage";
...
```

```
ReactDOM.render(
  <App>
    <VoteListPage />
  </App>,
  document.getElementById("root")
);
```

Wenn du den Client zu diesem Zeitpunkt im Browser öffnest, werden die Votes vom Server geladen. Die entsprechende Anfrage an den Server kannst du dir zum Beispiel in den Chrome Developer Tools ansehen (den aktuellen Stand findest du in `schritte/ 07a_loadvotes`).

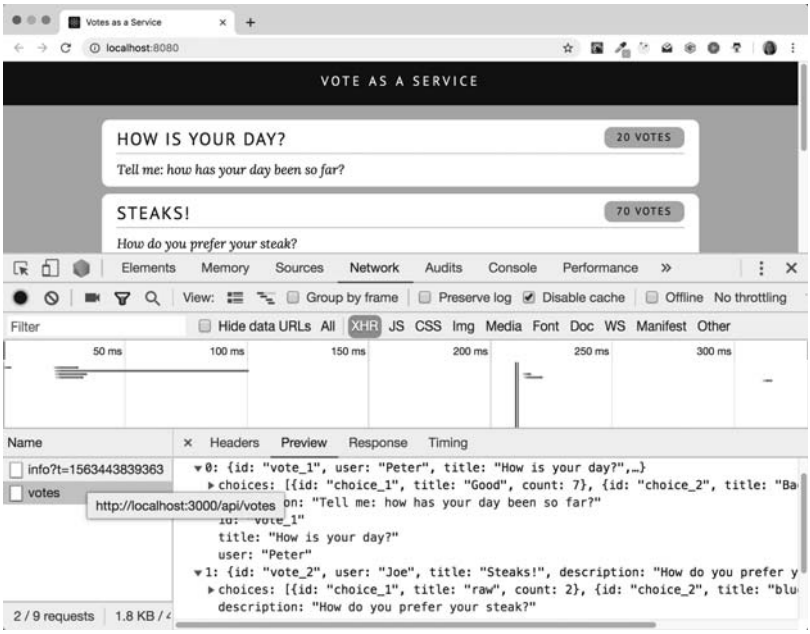

Abb. 7–4
Aufruf des API-Servers in den Chrome Dev Tools dargestellt

Im nächsten Schritt sorgen wir dafür, dass auch die abgegebenen Stimmen für eine Umfrage auf dem Server gespeichert werden.

Schritt 4: Speichern von neuen Umfragen und Stimmabgaben

Das Speichern von Daten passiert infolge eines Events (z.B. Anklicken einer Choice). Daraufhin haben wir bislang die im Client gespeicherte Liste aktualisiert. Nun soll aufgrund des Events ein Serveraufruf erfolgen, der die neue Umfrage speichert bzw. die Stimmabgabe registriert. Da wir uns in einem Event Handler nicht im Renderzyklus befinden, dürfen wir hier Seiteneffekte verwenden, und somit über die API auf den Server zugreifen. Die beiden Event Handler in der `VoteListPage` heißen `addVote` (neue Umfrage) bzw. `registerVote` (Stimmabgabe für

eine Umfrage). Beim Anlegen einer neuen Umfrage kommt die Umfrage als Antwort auf den HTTP-Aufruf vom Server zurück, sodass wir diese direkt in unseren Zustand einfügen können. Bei der Registrierung einer Stimmabgabe gibt uns der Server nur zurück, ob die Registrierung geklappt hat. Aus diesem Grund laden wir hier die Umfragen einfach komplett neu.

Die beiden Event Handler übergeben wir dann dem `VoteController`, den wir im nächsten Schritt leicht anpassen müssen.

Speichern von Daten in der VoteListPage

```
export default function VoteListPage() {
  const [allVotes, setAllVotes] = React.useState(null);

  async function loadVotes() { ... }

  React.useEffect( ... }

  async function registerVote(vote, choice) {
    await sendJson("PUT",
      `/api/votes/${vote.id}/choices/${choice.id}/vote`
    );

    loadVotes();
  }

  async function addVote(vote) {
    const newVote = await sendJson("POST", "/api/votes", vote);
    setAllVotes( currentVotes => [...currentVotes, newVote]);
  }

  if (!allVotes) { ... }

  return (
    <VoteController
      votes={allVotes}
      onRegisterVote={registerVote}
      onSaveVote={addVote}
    />
  );
}
```

Wenn du dir die `addVote`-Funktion ansiehst, erkennst du, dass dort mittels eines POST-Requests die übergebene Umfrage auf dem Server gespeichert und als Ergebnis auch zurückgeliefert wird. Das Ergebnis enthält dann auch die fehlenden Ids, die wir im vorherigen Kapitel noch improvisiert vergeben haben. Die neue Vote wird dann einfach im `allVotes`-State hinzugefügt, so wie das zuvor im `VoteController` passiert ist. Eine Änderung haben wir hier allerdings vorgenommen: Der `setAllVotes`-Funktion wird nicht die Liste der neuen Umfragen übergeben, sondern eine Callback-Funktion. Diese Callback-Funktion ruft

React mit dem jeweils aktuellen Wert des Zustands auf. Der Rückgabewert der Callback-Funktion wird dann von React als neuer Zustand gesetzt.

Nebenläufigkeiten und Zustand

Hintergrund ist, dass der Event Handler (`addVote`) immer nur Zugriff auf den State hat, so wie dieser zu dem Zeitpunkt aussah, als der Event Handler aufgerufen wurde. Im `VoteController` war das auch kein Problem, weil der Event Handler aufgerufen wurde und die Liste im State unmittelbar angepasst hat. In der neuen Variante ist `addVote` aber asynchron. Zwischen dem Senden und Empfangen der Daten vom Server und dem Aufruf von `setAllVotes` kann Zeit vergehen und der State kann sich somit in der Zwischenzeit verändert haben:

Nebenläufigkeit beim Laden von Daten

1. Eine neue Umfrage *U1* wird angelegt und `addVote` wird ausgeführt.
2. Server-Request *S1* startet, um die Umfrage zu speichern; `addVote` »pausiert« so lange.
3. In der Zwischenzeit wird von der Benutzerin eine neue Umfrage *U2* angelegt (die UI ist durch das Warten auf den Server-Request *S1* nicht blockiert!) und `addVote` wird ausgeführt.
4. Server-Request *S2* startet, um die Umfrage *U2* zu speichern.
5. Server-Request *S2* kommt zurück (Server-Request *S1* läuft immer noch) und in `addVote` wird die Liste der Umfragen um die neue Umfrage *U2* erweitert und in den Zustand gesetzt.
6. Nun kommt Server-Request *S1* zurück mit der Umfrage *U1* vom Server. Der Event Handler sieht allerdings noch den `allVotes`-Zustand vom Zeitpunkt seines Aufrufs (Schritt 1) und nicht die in Schritt 5 um *U2* erweiterte Liste. Wenn der Event Handler nun ebenfalls die Liste modifiziert und in den Zustand setzt, ist die Änderung aus Schritt 5 überschrieben.

Durch die Verwendung der Callback-Funktion kann dieses Szenario nicht eintreten, da der Callback-Funktion in Schritt 6 der aktuelle Wert aus dem Zustand übergeben wird (also Stand nach Schritt 5), sodass das Hinzufügen der Umfrage *U1* nicht dazu führt, dass die Umfrage *U2* »gelöscht« wird.

Ob diese Konstellation in deiner Anwendung vorkommt, hängt von mehreren Faktoren ab. Wäre zum Beispiel die UI während des Serverzugriffs blockiert, z.B. mit einer »Bitte warten«-Meldung, kann das Problem so nicht auftreten. Aber du solltest dir zumindest im Klaren darüber sein, dass durch das Arbeiten mit asynchronen APIs grundsätzlich solche Probleme entstehen können.

Den `VoteController` müssen wir noch anpassen, damit er unsere beiden Event Handler auch aufruft. Dazu fügen wir im `VoteController` die Properties `onRegisterVote` und `onSaveVote` hinzu, die Callback-Funktionen mit Event Handlern erwarten. Über diese beiden Properties können wir dann die Event Handler aus der `VoteLoadPage` (`registerVote` und `addVote`) übergeben. Bei den entsprechenden Ereignissen führt der `VoteController` diese dann aus. Da der `VoteController` die Votes nun selbst nicht mehr verändert, sondern nur noch darstellt, können wir dort auch den `allVotes`-State entfernen.

```
function VoteController({ votes, onSaveVote, onRegisterVote }) {

  // allVotes-State entfällt hier
  ...

  function saveVote(vote) {
    closeVoteComposer();

    // onSaveVote ist jetzt Event Handler aus VoteListPage
    onSaveVote(vote);
  }

  return (
    <div>
      { /* onRegisterVote ist Event Handler aus VoteListPage */ }
      <VoteList
        allVotes={votes}
        currentVoteId={currentVoteId}
        onSelectVote={setCurrentVote}
        onDismissVote={unsetCurrentVote}
        onRegisterVote={onRegisterVote}
      />
      ...
    </div>
  );
}
```

Diesen Stand findest du im Verzeichnis `schritte/07b_load_and_save_votes`.

7.2 Seiteneffekte mit useEffect

Das Rendern einer Komponente muss seiteneffektfrei sein. Das bedeutet, dass innerhalb der Komponentenfunktion beispielsweise nicht mit dem nativen DOM gearbeitet werden darf und auch keine Serverzugriffe erfolgen dürfen. Für Seiteneffekte kommt der `useEffect`-Hook zum Einsatz.

D Einführung in TypeScript

In diesem Anhang geben wir dir eine kurze Einführung in die Sprache TypeScript, die unter anderem ein Typsystem für JavaScript-Code zur Verfügung stellt und sehr gute Unterstützung für den Einsatz in React-Anwendungen bietet.

D.1 Motivation

JavaScript ist eine dynamisch getypte Sprache. Das bedeutet, dass einer Variablen nicht nur jederzeit ein anderer Wert zugewiesen werden kann, sondern dass dieser Wert auch von unterschiedlichen Typen sein kann. Eine Variable `name` kann somit beispielsweise zunächst vom Typ `string` sein und danach zum Typ `number` oder sogar zu einer Funktion werden:

```
let name = "Klaus";      // typeof name === string
name = 77;               // typeof name === number
name = function greet() { return "hello" }
                         // typeof name === function
```

Während die dynamische Typisierung auf der einen Seite sehr praktisch ist, weil wir uns keine Gedanken um die Typen machen müssen, kann sie auf der anderen Seite fehleranfällig und schwer verständlich sein.

Sehen wir uns als Beispiel die Signatur einer Funktion an:

```
function sayHello(person) { ... };
```

Ohne in den Code zu schauen, wissen wir nicht, was `person` sein soll. Ein String? Ein Objekt? Darf der Parameter `null` sein oder gar ganz weggelassen werden? Weiteres Problem: Was mag die Funktion zurückgeben? Auch das ist nicht ersichtlich. Ohne Dokumentation bleibt uns nur der Blick in den Quellcode.

Wenn wir an unsere React-Anwendung denken, haben wir es mit ähnlichen Problemen zu tun. Nehmen wir beispielsweise den State einer Komponente:

```
const [name, setName] = React.useState("");
```

Hier können wir zumindest davon ausgehen, dass der State ein String sein soll. Aber dürfte er auch `null` annehmen? Kann die Anwendung damit umgehen?

Das gleiche Problem gibt es mit den Properties einer Komponente:

```
function Greeter(props) { ... }
```

Von außen betrachtet können wir nicht erkennen, ob und welche Properties diese Komponente erwartet. Aber auch hier ist das Problem, dass ein fehlerhafter Aufruf der Komponente (zu wenig oder falsche Properties) erst zur Laufzeit bemerkt wird und dann auch nur, wenn die Anwendung den fehlerhaften Zustand überhaupt durchläuft, das heißt potenziell nur in sehr wenigen Situationen, die schwer zu finden bzw. zu reproduzieren sind. Darüber hinaus sind Properties read-only. Dennoch würde folgender Code »funktionieren«, also ausgeführt werden, wenngleich es zur Laufzeit zu einem Fehler kommt:

```
function Greeter(props) {
  props.name = ""; // Verboten: Properties sind read-only!
}
```

Linter

Die Probleme, die durch die dynamische Typisierung auftreten, können auf zwei Wegen gelöst werden: durch statische Codeanalyse (Linter) oder durch Type Checker. Ein Linter wie ESLint kann durch statische Codeanalyse zumindest einige der genannten Probleme feststellen, zum Beispiel wenn eine Variable verwendet, aber nicht zuvor definiert wurde:

```
function greet(name) {
  return `Hello, ${firstname}` // firstname not defined
}
```

Hier würde der Linter erkennen, dass `firstname` nicht definiert ist (ein möglicher Programmierfehler, vermutlich ist `name` gemeint).

Allerdings kann der Linter nicht feststellen, ob die Funktion `greet` korrekt aufgerufen wurde, da der Linter zwar weiß, dass es einen Parameter `name` gibt, aber nicht von welchem Typ dieser sein soll und ob er `null` annehmen kann oder gar weggelassen werden darf.

Type Checker

Ein konsequenterer Ansatz zur Vermeidung dieser Probleme ist ein statischer Type Checker. Hier gibt es für JavaScript zwei Werkzeuge, die beide auch mit React, also JSX-Code, umgehen können: Flow und

TypeScript. Flow ist ein reiner Type Checker, der von Facebook entwickelt und auch für den React-Code selbst verwendet wird.

TypeScript

TypeScript ist eine von Microsoft entwickelte Sprache, die auf JavaScript aufbaut. Aufgrund der hohen Verbreitung von TypeScript zeigen wir dir in diesem Anhang, wie TypeScript funktioniert und wie du es in deiner JavaScript- bzw. React-Anwendung einsetzen kannst.

Unabhängig davon, welchen Type Checker, Flow oder TypeScript, du verwendest, werden dir von diesem bereits zur Entwicklungszeit, also zum Beispiel beim Kompilieren deiner Anwendung durch Webpack, oder sogar schon in der Entwicklungsumgebung Fehler angezeigt. Du kannst sie also vermeiden bzw. beheben, bevor du deine Anwendung zum Testen ausführst.

Abb. D–1
TypeScript in Visual Studio Code

Im Folgenden geben wir dir eine kurze, von React unabhängige Einführung in die Sprache TypeScript. Eine ausführlichere Beschreibung findest du unter anderem in dem Online-Handbuch »Deep Dive«-TypeScript[1].

D.2 Die Sprache TypeScript

TypeScript ist eine von Microsoft entwickelte Sprache, die auf JavaScript aufbaut. TypeScript ist dabei eine Obermenge von JavaScript, sodass prinzipiell jeder gültige JavaScript-Code auch gültiger TypeScript-Code ist. Allerdings erweitert TypeScript die Sprache JavaScript um einige Features. Das aus unserer Sicht wichtigste Feature ist der

1 *https://basarat.gitbooks.io/typescript/content/docs/getting-started.html*

statische Type Checker, dazu gleich mehr. Daneben gibt es aber beispielsweise auch Aufzählungstypen, die in JavaScript nicht existieren (Enums), sowie Sichtbarkeiten in Klassen (Methoden und Properites einer Klasse können in TypeScript als `private` oder `protected` deklariert sein und sind dann entweder gar nicht außerhalb der Klasse oder nur in Unterklassen sichtbar).

Zielversionen

Da durch die Spracherweiterung der TypeScript-Code nicht direkt vom Browser ausgeführt werden kann, bringt TypeScript einen Compiler mit, der den Sourcecode (ähnlich wie Babel) in verschiedene JavaScript-Versionen zurückübersetzen kann. Dabei wird geprüft, ob alle Typen korrekt verwendet werden und die proprietäre Syntax wird entfernt. Da der Compiler auch in der Lage ist, JSX-Code in JavaScript zu übersetzen, kann beim Einsatz von TypeScript in React-Projekten auf Babel grundsätzlich verzichtet werden (Create React App verwendet Babel zusätzlich wegen weiterer Plug-ins).

Das wesentliche Feature von TypeScript ist dessen statisches Typsystem, mit dem eine JavaScript-Anwendung typisiert werden kann.

D.3 Grundlagen des TypeScript-Typsystems

Type Inference

Typen in TypeScript können entweder eingebaute Typen wie `string`, `number`, `boolean` oder ein Array sein oder auch eigene Typen. Die Festlegung von Typen kann explizit oder implizit erfolgen. Dort wo keine Typen explizit angegeben werden, versucht TypeScript die gültigen Typen zu ermitteln (»Type Inference«). Wo das nicht möglich ist, müssen die Typen von dir angegeben werden. An allen anderen Stellen ist es optional. Das führt in der Praxis dazu, dass an nur erstaunlich wenigen Stellen überhaupt zwingend Typangaben erforderlich sind.

Im folgenden Beispiel »weiß« TypeScript, dass `name` vom Typ `string` ist, da der Variablen beim Initialisieren ein String zugewiesen wird. Die darauffolgende Zuweisung einer Zahl an die Variable verbietet TypeScript aus diesem Grund:

Beispiel: Type Inference

```
let name = "Klaus"
name = 77; // Type '77' is not assignable to type 'string'
```

Alternativ kannst du eine Typangabe explizit angeben. Dazu schreibst du durch einen Doppelpunkt getrennt den Typ hinter den Namen der Variablen:

```
let name: string = "Klaus";
name = 77; // ERROR: Type '77' is not assignable to type 'string'
```

Funktionsargumente

Bei Funktionen musst du zwingend für alle Argumente einen Typ angeben, da TypeScript die Typen hier nicht selbst ableiten kann. Auch hier werden die Typangaben mit Doppelpunkt getrennt hinter die einzelnen Funktionsparameter geschrieben. Den Rückgabewert wiederum kann TypeScript selbst herleiten. So erkennt TypeScript im folgenden Beispiel, dass der Rückgabewert vom Typ `string` ist:

```
function sayHello(name: string) {
  return `Hello, ${name.toUpperCase()}`;
}

// Alternativ als Arrow-Funktion:
const sayHello = (name: string) => `Hello, ${name.toUpperCase()}`;

let greet = sayHello("Klaus");

greet = 99;   // ERROR: Type '99' is not assignable
              // to type 'string'

sayHello(77); // ERROR: Argument of type '77' is not assignable
              // to parameter of type 'string'.
```

Einer Variablen oder einem Parameter können in TypeScript auch mehr als ein Typ zugewiesen werden. Das ist einerseits notwendig, weil JavaScript solche Konstrukte grundsätzlich zulässt, und andererseits ist auch `null` und `undefined` in TypeScript ein eigener Typ. Aus diesem Grund akzeptiert die oben gezeigte `sayHello`-Funktion auch kein `null` als Parameter. Auch kann der Parameter nicht ganz weggelassen werden. Beides würde zu einem Compile-Fehler führen. Daraus ergibt sich auch, dass TypeScript sicher sagen kann, dass der Aufruf von `toUpperCase` in jedem Fall funktioniert, denn `name` ist immer ein `string`, niemals `null` oder `undefined` oder von einem anderen Typ.

Union Types

Möchtest du einer Variablen oder einem Parameter mehr als einen Typ zuweisen, kannst du mehrere Typen mit | separiert hinschreiben (das ist dann ein »Union Type«). Wenn du ausdrücken möchtest, dass ein Parameter optional ist, kannst du ihn mit einem Fragezeichen markieren:

```
export function sayHello(name?: string | null) {
  return `Hello, ${name.toUpperCase()}`;
}

sayHello("Klaus") // OK
sayHello(null); // OK
sayHello(); // OK
sayHello().toUpperCase(); // OK: sayHello liefert string zurück
```

Nun wird allerdings die Funktion selbst nicht mehr kompilieren, da `toUpperCase` auf `name` aufgerufen wird, und `name` kann ja nun auch `null` oder `undefined` sein.

Du musst die Funktion also entsprechend anpassen, zum Beispiel:

```
function sayHello(name?: string | null) {
  if (name === null || name === undefined) {
    return null;
  }
  return `Hello, ${name.toUpperCase()}`;
}
```

In dieser Implementierung liefert die Funktion nun `null` zurück, wenn `name` weggelassen oder `null` übergeben wurde. Dadurch verändert sich allerdings auch der Rückgabetyp der Funktion! Die Funktion gibt nun einen `string` oder `null` zurück:

```
function sayHello(name?: string | null) { ... }

sayHello("Klaus").toUpperCase(); // ERROR: Object is
                                 // possibly 'null'
```

Rückgabewerte von Funktionen

Wie du siehst, haben wir durch die Änderung der Funktion ein Problem beim Verwender der Funktion erzeugt. Dieser konnte in der vorhergehenden Variante davon ausgehen, immer einen `string` zu erhalten, nun kann plötzlich auch `null` zurückkommen. Dieses Verhalten kann so gewollt sein, vielleicht ist es aber auch nur versehentlich entstanden und die Funktion sollte eigentlich weiterhin einen String zurückliefern. Um solche Fälle zu verhindern, kannst du bei Methoden explizit einen Rückgabewert angeben. In diesem Fall wird der Fehler (dass nun `null` zurückgegeben wird) in der Funktion angezeigt und nicht mehr beim Aufrufer. Das ist insbesondere bei Funktionen, die Teil einer öffentlichen Schnittstelle sind, sinnvoll, um zu vermeiden, dass eine Funktion »versehentlich« inkompatibel zu einer vorangegangenen Version wird:

Beispiel: Typen für Rückgabewerte

```
function sayHello(name?: string | null): string {
  if (name === null || name === undefined) {
    return null; // ERROR: Type 'null' is not assignable
                 // to type 'string'
  }
  return `Hello, ${name.toUpperCase()}`;
}

// Alternativ als Arrow-Funktion:
const sayHello = (name?: string | null): string => { ... }

// Verwendung
sayHello("Klaus").toUpperCase(); // Kein Fehler
```

Um den Fehler in der Funktion zu beseitigen, könnten wir nun beispielweise statt `null` einen Leerstring zurückgeben.

In unserem Fall möchten wir aber dabeibleiben, dass die Funktion auch `null` zurückliefern kann. Das bedeutet aber, dass der Aufrufer angepasst werden muss und explizit auf `null` prüfen muss:

```
function sayHello(name?: string | null) {
  if (name === null || name === undefined) {
    return null;
  }
  return `Hello, ${name.toUpperCase()}`;
}

const hello = sayHello("Klaus");

if (hello !== null) {
 const greet = hello.toUpperCase(); // OK
 // ...
}
```

Durch diese Prüfung weiß TypeScript, dass im `if`-Block der Typ von `hello` nur noch `string` sein kann. Der andere erlaubte Typ (`null`) wurde durch die `if`-Abfrage ausgeschlossen.

Type Narrowing

Dieses Feature nennt sich »Type Narrowing«, da die Typen einer Variablen durch entsprechende Abfragen »eingeengt« bzw. eingeschränkt werden. Das funktioniert nicht nur mit Prüfungen auf `null`, sondern auch mit anderen Typen. Schau dir dazu das folgende Beispiel an, in dem die `sayHello`-Funktion nun auch eine `number` als Parameter akzeptiert. Um das Beispiel möglichst verständlich zu machen, ist es etwas ausführlicher implementiert, als es eigentlich sein müsste (`null`- und `typeof`-Check könnten auch zusammengelegt werden):

Beispiel: Type Narrowing

```
function sayHello(name: string | number | null) {

  // Der Typ von name ist hier string, number oder null
  if (typeof name === "number") {

   // Der Typ von name ist hier number
    return null;
  }

// Der Typ von name ist hier nur noch name oder null
  if (name === null) {
    // name ist hier null
    return null;
  }

  // Der Typ von name ist hier nur noch string
  return `Hello, ${name.toUpperCase()}`;
}
```

Du siehst, wie sich der Typ in den Verzweigungen der Funktion »einschränkt«.

In den Beispielen oben, in denen wir zwischen einem konkreten Typ (`string` oder `number`) und `null` und/oder `undefined` unterscheiden wollten, haben wir immer eine explizite Typabfrage gemacht:

```
name === undefined || name === null
```

Eine häufige Fehlerquelle ist es, stattdessen den logischen Nicht-Operator (`!`) zu verwenden. In diesem Fall würde aber durch die implizite Typkonvertierung in JavaScript beispielsweise auch ein Leerstring oder eine `0` zu `true` ausgewertet werden, wie das folgende Beispiel zeigt:

```
function sayHello(name: string | number | null) {

  if (!name) {

   // Der Typ von name ist hier immer noch string, number oder
   // null
    return null;
  }

  // Hier ist name nur noch string oder number

  ...

}
```

Typinformationen nur zur Build-Zeit

Keine Laufzeitinformationen

Die Typinformationen in deiner Anwendung werden vom TypeScript-Compiler beim Build und in der IDE überprüft und dir werden entsprechende Fehler ausgegeben.

Beim Kompilieren werden allerdings die Typangaben vom Compiler vollständig entfernt, sodass diese nicht im erzeugten JavaScript-Code vorhanden sind. Damit stehen die Typinformationen in keiner Form zur Laufzeit zur Verfügung. Eine Art Reflection-API, wie beispielsweise aus Java oder C# bekannt, gibt es in TypeScript nicht. Typinformationen werden in TypeScript ausschließlich zur Build-Zeit verwendet.

D.3.1 Eigene Typen definieren

In TypeScript kannst du eigene Typen definieren, um Strukturen von Objekten zu beschreiben. Dies geschieht über die Schlüsselwörter `type` oder `interface`. Da Semantik, Funktionsweise und Funktionsumfang der beiden Schlüsselwörter mittlerweile fast identisch sind, verwen-

den wir in diesem Buch und in unserer Beispielanwendung ausschließlich `type`[2].

Damit kannst du erzwingen, dass eine Variable oder ein Parameter ein Objekt erhält, das genauso wie beschrieben vorliegt. Dazu gibst du die Namen der Objekteigenschaften sowie deren Typen an:

```
type Person = {
  firstName: string;
  age: number;
}
```

Auch hier gilt, dass die Eigenschaften des Objekts nicht `null` oder `undefined` sein dürfen, solange das nicht explizit angegeben ist:

```
const klaus:Person = { firstName: "Klaus", age: 32 };
const susi:Person = { firstName: "susi",  age: null}; // ERROR: Type
'null' is not assignable to type 'number'
```

Beim Verwenden eines Objekts prüft TypeScript, ob das Objekt auf den erwarteten Typ passt. So musst du zum Beispiel bei der Deklaration der Variablen nicht explizit hinschreiben, dass die Variable vom Typ `Person` ist, du kannst sie aber trotzdem an eine Funktion übergeben, die eine `Person` erwartet, oder einer Variablen zuweisen, die explizit eine `Person` erwartet. Auch wenn der Rückgabewert einer Funktion der erwarteten Struktur eines Typs entspricht, kannst du diesen einer Variablen vom entsprechenden Typ zuweisen:

```
function greet(p: Person) {
  const g = `Hello, ${p.name}`; // ERROR: Property 'name' does not
                                // exist on type 'Person'

  return `Hello, ${p.firstName}`;  // OK
}

const maja = { firstName: "Maja", age: 40 };

// Zuweisung an Variablen
const majaThePerson: Person = maja;

// Verwendung der Funktion
greet(maja);

// Direkte Verwendung ohne Typangabe
greet({
  firstName: "Karl",
  age: 42
```

2 Eine Beschreibung der Unterschiede zwischen type und interface findest du im neuen TypeScript-Handbuch, das gerade geschrieben wird: *https://microsoft.github.io/TypeScript-New-Handbook/chapters/everyday-types/#interface-vs-alias*.

```
});

// createPerson liefert ein Objekt zurück
function createPerson(name: string, alter: number) {
  return { name: firstName, age: alter };
}

// Struktur des Rückgabewerts von createPerson
// passt auf Person
const cathy: Person = createPerson("Cathy", 35);
```

Arrays

Auch Arrays lassen sich mit TypeScript typsicher beschreiben. Dazu kannst du den Typnamen, der ein Array enthalten soll, gefolgt von einem leeren Paar eckiger Klammern hinschreiben oder die generische Schreibweise verwenden:

Beispiel: Arrays

```
function greetAll(persons: Person[]){
  return persons.map(person => {
    // Typ von "person" ist hier "Person"
    const lastName = person.lastName;
     // ERROR: Property 'lastName' does not exist on type 'Person'

    return person.firstName; // OK
  });
}

// Alternativ:
function greetAll(persons: Array<Person>) { ... }

const klaus: Person = { firstName: "Klaus", age: 32 };
const susi = { firstName: "Susi", age: 34 };

// Alle drei Parameter sind strukturidentisch zu Person
const greetings = greetAll([
  klaus,
  susi,
  {
    firstName: "Karl",
    age: 42
  }
]);

console.log(greetings[1]); // "Susi"
console.log(greetings.toUpperCase()); // FEHLER: Property
'toUpperCase' does not exist on type 'string[]'
```

Im Beispiel oben siehst du, wie TypeScript den Typ auch in der `map`-Funktion automatisch erkennt und die korrekte Verwendung sicherstellen kann. Auch der Rückgabetyp der `greetAll`-Funktion (ein Array von Strings) wird korrekt erkannt.

Funktionen in Objekten

Da Objekte in JavaScript auch Funktionen aufnehmen können, kannst du auch Funktionen mitsamt ihren erwarteten Parametern angeben. Die Methodenparameter gibst du dabei in Klammern an, den Rückgabewert mit einem Pfeil dahinter:

Beispiel: Typ-Angaben für Funktionen

```
type Person = {
  name: string;
  updateName: (name: string) => string;
}

const klaus: Person = {
  name: "Klaus",
  updateName: (name: string) => { console.log(name) } // OK
};

const wrong: Person = {
  name: "Klaus",
  updateName: (name: boolean) => { ... }
      // ERROR: Types of parameters 'name' and 'name'
      // are incompatible.
};
```

Typen erweitern

Typen können erweitert werden. Dazu kannst du mehrere Typen mit dem &-Operator verknüpfen (»Intersection Types«):

```
type Employee = Person & { salary: number; }

const klaus:Person = {
  firstName: "Klaus", age: 37, salary: 60000
};
```

Du kannst übrigens überall dort, wo du benannte Typen hinschreibst, auch direkt die Typdefinition angeben, wenn das für dich einfacher ist:

```
function greetAll(persons: { firstName: string, age: number }[]):
string[] { ... }
```

Und natürlich kannst du auch Destructuring bei Parametern durchführen. In dem Fall musst du den Typ hinter die geschweifte Klammer schreiben:

```
function greet({firstName}: Person) { ... }
```

Type Aliase

Außerdem kannst du mit type Aliase für bestimmte Typen vergeben, um zum Beispiel wiederkehrende Konstrukte zu vereinfachen oder Dinge fachlich zu beschreiben:

```
type Greetable = string | null;
function sayHello(name: Greetable) { ... }

type PrimaryKey = string;
function loadVote(id: PrimaryKey) { ... }
```

Typen exportieren

Genau wie Klassen, Funktion etc. in JavaScript kannst du in TypeScript auch Typen aus einem Modul exportieren, um den Typ in anderen Modulen verwenden zu können. Somit könntest du zum Beispiel ein Modul mit fachlichen Typen anlegen, das du dann innerhalb deiner Anwendung verwenden kannst:

```
// domain.ts
export type Person = { ... }
export type Employee = { ... }

// greeter.js
import { Person } from "./domain";

export function sayHello(person: Person) { ... }
```

D.3.2 Klassen in TypeScript

Neben den gezeigten Typen lassen sich mit TypeScript auch Klassen verwenden. Hierbei ist die Syntax zunächst identisch mit der Syntax der ES6-Klassen. Neu ist allerdings, dass du sämtliche Felder der Klasse sowie die Parameter der Methoden zwingend beschreiben musst. Dabei kannst du auch angeben, ob eine Variable oder eine Methode *protected* oder *private* sein soll. Protected Member sind nur in der Klasse selbst und in allen Unterklassen sichtbar, private Member nur in der Klasse selbst. Wenn du keine Sichtbarkeit hinschreibst, ist der Member überall sichtbar (public), so wie in JavaScript auch:

```
class Greeter {
  // private ist nur innerhalb Greeter sichtbar
  private phrase: string;

  constructor(phrase: string) {
    this.phrase = phrase;

    this.name = "Klaus"; // Property 'name' does not
                         // exist on type 'Greeter'
  }

  // greet ist public und von überall aus aufrufbar
  greet(name: string) {
    const greeting = `${phrase}, ${name}`; // ERROR: Cannot find
         // name 'phrase'. Did you mean the instance member
         //'this.phrase'

    return `${this.phrase}, ${name}`; // OK
  }
}

const wrong = new Greeter(); // ERROR: Expected 1 arguments,
                             // but got 0.
```

```
const g = new Greeter("Hello");
console.log(g.greet("Susi")); // Hello, Susi

const greetingPhrase = g.phrase; // ERROR Property 'phrase' is
                                 // private

g.phrase = "Moin"; // ERROR Property 'phrase' is private
```

D.3.3 Der any-Type

Bei der Arbeit mit TypeScript kann es dir passieren, dass du an Stellen gelangst, an denen du keine Typdefinition hinschreiben kannst oder willst, insbesondere wenn du mit Legacy-Code arbeitest, der (noch) in JavaScript geschrieben ist. In solchen Fällen kannst du den Typ `any` verwenden. Dieser schaltet die Typüberprüfung praktisch aus, sodass du an eine Variable bzw. einen Parameter, der vom Typ `any` ist, alle Typen zuweisen kannst und auch mit diesem Typen dann alles machen kannst. Das sollte wirklich nur der letzte Ausweg sein, da du damit die Vorteile von TypeScript verwirfst.

Beispiel: Verwendung von any

```
function sayHello(name: any) {
  name++; // OK
  name = true; // OK
  return name.toUpperCase(); // OK
}

sayHello(null); // OK
sayHello(„Klaus"); // OK
sayHello(123); // OK
```

Für den gezeigten Code wird TypeScript keine Compiler-Fehler ausgeben, trotzdem wird der Code natürlich zur Laufzeit mutmaßlich nicht funktionieren ...

Implizites Zuweisen von any

Wenn du einen Typ nicht explizit angibst und TypeScript es nicht schafft, den Typ herzuleiten (z.B. bei einem Parameter einer Funktion), weist TypeScript diesem Typ automatisch (implizit) den Typ `any` zu. Das implizite Zuweisen von `any` kann aber per Konfiguration verboten werden (damit das nicht »aus Versehen« passiert und versehentlich die Typprüfung abgeschaltet wird). Im Workspace unserer Beispielanwendung ist die implizite Zuweisung verboten und auch in Projekten, die neu mit Create React App erzeugt werden, ist diese Konfiguration aktiv:

```
function sayHello(name) { // ERROR: Parameter 'name' implicitly
                          // has an 'any' type.
 ...
}

function sayHello(name: any) { // OK
  ...
}
```

D.4 Externe Typbeschreibungen

Im vorherigen Kapitel haben wir dir gezeigt, wie du Typdefinitionen für deinen bestehenden JavaScript-Code schreiben kannst, damit der TypeScript-Compiler ihn auf korrekte Verwendung hin überprüfen kann.

Für bestehenden JavaScript-Code, den du nicht selbst geschrieben hast, also für von dir verwendete externe Module, wie beispielsweise React, wäre es sehr lästig, wenn du dafür ebenfalls die Typdefinitionen schreiben müsstest. Glücklicherweise enthalten mittlerweile einige in JavaScript geschriebene Module bereits auch TypeScript-Typdefinitionen (z.B. Redux). In diesem Fall musst du nichts weiter tun. TypeScript findet die Typdefinitionen automatisch im entsprechenden Verzeichnis unterhalb von `node_modules`. Alternativ stehen für die allermeisten JavaScript-Module externe Typdefinitionen zur Verfügung. Diese werden meist von der Community erstellt und gepflegt, häufig auch gemeinsam mit den Entwicklern der entsprechenden Bibliothek. Diese Typdefinitionen werden auf der Plattform DefinitelyTyped[3] zentral gesammelt. Du kannst sie wie ein gewöhnliches `npm`-Modul mittels `npm install` zu deinem Projekt hinzufügen. Üblicherweise heißen die Typmodule genauso wie auch das Modul, das sie beschreiben, nur mit dem `npm`-Scope `@types`. So kannst du beispielsweise die Typdefinitionen für React und React-DOM hinzufügen:

```
npm install --save @types/react @types/react-dom
```

Wenn du Create React App verwendest, sind diese Typdefinitionen für dich bereits hinzugefügt. Analog kannst du aber auch beispielsweise die Typen für den React Router oder React Redux hinzufügen.

Während dieses Vorgehen in der Regel sehr praktisch und bequem ist, kann es zu Problemen hinsichtlich der Versionierung kommen. So müssen die Version von TypeScript, die Version des Moduls und die Version des Typmoduls zusammenpassen. Natürlich muss eine Änderung an einem der drei Artefakte nicht notwendigerweise dazu führen, dass die anderen Artefakte nicht mehr dazu passen, aber das kann passieren. Aus diesem Grund ist es manchmal ratsam, insbesondere beim Erscheinen einer neuen TypeScript- oder React-Version, zunächst ein paar Tage abzuwarten, bis die Typdefinitionen angepasst wurden, und dann erst umzustellen. Das wird in der Regel relativ zügig passieren und dann solltest du auch nach Möglichkeit die Versionen in deinem Projekt aktualisieren, um zu vermeiden, dass du später große Versionssprünge mit vielen Anpassungen (und entsprechend komplexer Fehlersuche) vornehmen musst.

---

3 *http://definitelytyped.org/*

Eigene Typdeklarationen

Wenn es keine fertigen Typdeklarationen für eine Bibliothek gibt, kannst du die Typdeklaration auch selbst schreiben. Dazu musst du eine sogenannte Typdeklarationsdatei anlegen (diese endet mit `.d.ts`), in der du ein Modul deklarierst. Wie das funktioniert, ist im Detail in der TypeScript-Dokumentation beschrieben.