Nils Hartmann · Oliver Zeigermann

# React

Grundlagen, fortgeschrittene Techniken und Praxistipps – mit TypeScript und Redux

dpunkt.verlag

# Inhalt

**Cover**

**Über den Autor**

**Titel**

**Impressum**

**Inhaltsverzeichnis**

**Teil I Einstieg**

1 Einleitung

1.1 Was ist React?

1.2 Warum React?

1.3 Beziehung zu anderen Technologien

1.4 Vergleich mit anderen Webtechnologien

1.5 Wie man dieses Buch benutzt

1.6 Voraussetzungen für dieses Buch

1.7 Änderungen gegenüber der ersten Auflage

1.8 Website zum Buch

1.9 Danksagungen

2 Schnelldurchgang – React im Überblick

2.1 Zusammengefasst: Unterschiede zwischen Hooks- und Klassen-API

2.2 Ein React-Projekt beginnen mit Create React App

2.3 Zusammenfassung

3 Die Beispielanwendung: »Vote as a Service«

3.1 Die Beispielanwendung installieren und ausführen

3.2 Fachliches Modell

3.3 Die Anwendung schrittweise entwickeln

3.4 Zusammenfassung

**Teil II React**

4 Eine React-Komponente

4.1 Hands-on: eine Komponente

4.2 React-Komponenten in der Übersicht

4.3 Hooks

4.4 Zustand verwalten mit useState

4.5 Ereignisse und Event Handler

4.6 JSX zur Beschreibung der UI

4.7 Rückgabewerte von Komponenten

4.8 Einhängen der Anwendung in den DOM

4.9 Arbeiten mit CSS

4.10 Zusammenfassung

5 Arbeiten mit Komponentenhierarchien

5.1 Hands-on: Hierarchien von Komponenten

5.2 Kommunikation zwischen Komponenten

5.3 Das »Render Properties«-Pattern

5.4 Performance-Optimierung: Caching von Werten mit useMemo

5.5 Performance-Optimierung: Rendern vom Komponenten unterdrücken

5.6 Performance-Optimierung: Code-Splitting mit React.lazy und Suspense

5.7 Der React Strict Mode

5.8 Zusammenfassung

6 Formulare mit React

6.1 Hands-on: ein Editor für Umfragen

6.2 Hintergrund: Databinding

6.3 Controlled Components

6.4 Uncontrolled Components

6.5 Auf native DOM-Elemente zugreifen: das ref-Property

6.6 Komplexen Zustand mit useReducer verwalten

6.7 Zusammenfassung

7 Arbeiten mit Seiteneffekten: asynchrone Serverzugriffe

7.1 Hands-on: Serveranbindung

7.2 Seiteneffekte mit useEffect

7.3 Code wiederverwenden mit Custom Hooks

7.4 Server-Requests mit useState und useReducer

7.5 Ausblick: Daten laden mit Suspense

7.6 Zusammenfassung

**Teil III Über React hinaus**

8 React-Anwendungen testen

8.1 Hands-on: Testen mit Jest und React Testing Library

8.2 Überblick: React-Anwendungen testen

8.3 React-Test-Bibliotheken

8.4 Snapshot Testing

8.5 Zusammenfassung

9 Der React Router

9.1 Hands-on: der React Router im Schnelldurchgang

9.2 Anpassungen an der Vote-Anwendung

9.3 Links und Routen

9.4 Navigation über die History

9.5 Authentifizierung

9.6 Die React Context API

9.7 Lazy Loading mit dem React Router

9.8 Testen

9.9 Zusammenfassung

10 Externes Statemanagement mit Redux

10.1 Motivation
10.2 Hands-on: eine Redux-Anwendung
10.3 Redux in der Vote-Anwendung
10.4 Arbeiten mit dem globalen Zustand
10.5 Asynchrone Actions
10.6 Alternative zu Hooks: die connect-Funktion
10.7 Integration von Redux und React Router
10.8 Testen von Redux-Anwendungen
10.9 Exkurs: Codestruktur von großen React-Anwendungen
10.10 Alternative zu Redux: MobX
10.11 Zusammenfassung
11 React-Anwendungen mit TypeScript
11.1 Hands-on: eine React-Anwendung mit TypeScript
11.2 TypeScript in React-Anwendungen
11.3 React-Komponenten mit TypeScript
11.4 Der React Router mit TypeScript
11.5 Redux
11.6 Zusammenfassung
12 GraphQL mit dem Apollo Client für React
12.1 Hands-on: ein GraphQL-Client
12.2 GraphQL in der Vote-Anwendung
12.3 Mutations
12.4 Der Apollo Client Cache
12.5 GraphQL für externes Statemanagement?
12.6 Der Apollo React Client mit TypeScript
12.7 Zusammenfassung
**Anhang**
A Serverseitiges Rendern mit React

A.1 Ein Beispiel

A.2 Gründe für serverseitiges Rendern

A.3 Serverseitiges Rendern im Überblick

A.4 Herausforderungen

A.5 Asynchrones Datenladen mit Redux und dem React Router

B Komponenten als Klassen

B.1 React-Komponenten als ES6-Klasse

B.2 Methoden-Binding für Event Handler

B.3 Zugriff auf native DOM-Elemente

B.4 Arbeiten mit Seiteneffekten

B.5 Beispiel: Error Boundaries

B.6 Klassenkomponenten mit TypeScript

C Einführung in ES.Next

C.1 Einleitung

C.2 Block-Scope

C.3 Template Strings

C.4 Destructuring

C.5 Klassen

C.6 Vererbung

C.7 Erweiterte Objekt-Literale

C.8 Module, Exporte und Importe

C.9 Arrow-Funktionen

C.10 Default- und Rest Parameter

C.11 Spread-Operator für Arrays

C.12 Object.assign und Spread-Operator bei Objekten

C.13 Promises

C.14 async/await

C.15 Generatorfunktionen

C.16 Die fetch-API
D Einführung in TypeScript
D.1 Motivation
D.2 Die Sprache TypeScript
D.3 Grundlagen des TypeScript-Typsystems
D.4 Externe Typbeschreibungen
E Übersicht GraphQL
E.1 Abfragen
E.2 Das Schema: Beschreibung der API
**Index**

# 4 Eine React-Komponente

In diesem Kapitel beschäftigen wir uns mit dem Kern von React: den Komponenten. Wir zeigen dir die wichtigsten Konzepte und Eigenschaften von React-Komponenten. Dabei konzentrieren wir uns zunächst auf eine einzelne, isolierte Komponente.

## 4.1 Hands-on: eine Komponente

Beginnen wir zunächst mit der Entwicklung einer sehr einfachen Komponente für die Vote-Anwendung. Dabei handelt es sich um die `Choice-Bar`-Komponente. Diese Komponente ist in der Anwendung dafür zuständig, eine einzelne Antwort (Choice) einer Umfrage (Vote) anzuzeigen.

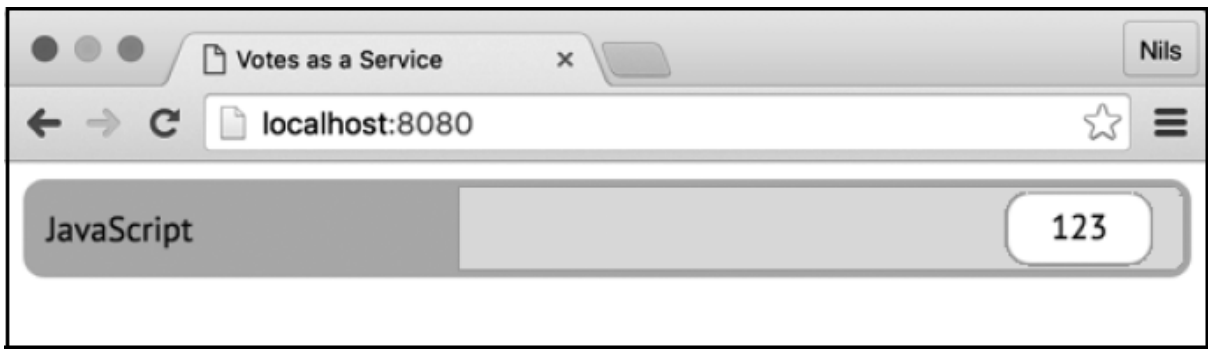


***Abb. 4–1*** *Die Komponente ChoiceBar*

Wir entwickeln diese Komponente nun Schritt für Schritt in einer zunächst sehr einfachen Variante.

### Schritt 1: Die Komponentenfunktion anlegen

Erinnerung: Wir gehen davon aus, dass du das Beispiel wie in Abschnitt 3.1 beschrieben im Verzeichnis `hands-on` des Vote-Example-Repositorys entwickelst.

Du solltest außerdem vorher in dem Verzeichnis `npm install` ausgeführt haben, damit die benötigten Bibliotheken und Build-Tools (insbesondere natürlich React und der Compiler Babel) lokal bei dir installiert sind.

Lege bitte im Verzeichnis `src/components` eine neue JavaScript-Datei mit dem Namen `ChoiceBar.js` an, die die Komponente enthalten wird.

In der neu angelegten Datei musst du zunächst React importieren. Das ist immer notwendig, wenn du in einer Datei JSX verwenden willst – auch dann, wenn du im Sourcecode gar keine direkten Referenzen auf das React-Modul hast.

*ES6-Importe und -Exporte erklären wir dir im Anhang.*

Da wir die `ChoiceBar`-Komponente als Funktion implementieren wollen, kannst du auch schon die Funktion dazu anlegen. Der Name der Funktion entspricht dabei dem Namen der Komponente. Da eigene Komponenten in React mit einem Großbuchstaben beginnen müssen, heißen sowohl unsere Komponente als auch die Funktion jeweils `ChoiceBar`. Um die Komponente außerhalb der `ChoiceBar.js`-Datei nutzen zu können, muss die Funktion mit dem Schlüsselwort `export` exportiert werden. Füge also bitte folgenden Code in die `ChoiceBar.js`-Datei ein:

```
import React from “react”;

export default function ChoiceBar() {

}
```

**Schritt 2: Die Komponentenfunktion implementieren**

Eine Komponentenfunktion liefert eine Baumstruktur von React-Elementen zurück, die zur Laufzeit zu HTML-Elementen gerendert werden. Dank der Spracherweiterung JSX können die Elemente direkt in den Code geschrieben werden.

Füge nun also folgenden Inhalt in die Funktion der Komponente ein, die zunächst noch sehr einfach ist und eine fiktive Auswahlmöglichkeit mit dem Titel »JavaScript« und einer Stimmenanzahl von 123 anzeigt:

```
import React from “react”;

export default function ChoiceBar() {

  return <div>
```

```
      <div>JavaScript</div>

      <div>123</div>

    </div>;

}
```

### Schritt 3: Die Komponente auf einer HTML-Seite einbinden

*DOM-Element für Anwendung auswählen*

Bevor wir die Komponente weiter ausbauen, wollen wir sie zunächst in unsere Seite einbinden. Dazu musst du React ein Element auf deiner HTML-Seite angeben, unterhalb dessen deine Komponente eingefügt werden soll. Alle Elemente, die sich unter dem ausgewählten Element befinden, werden von React entfernt und durch die HTML-Elemente deiner Komponente ersetzt. In der React-Dokumentation wird übrigens davon abgeraten, das `body`-Element zu verwenden, da sich dort – zum Beispiel durch Drittbibliotheken hinzugefügt – weitere DOM-Elemente befinden könnten, die von React dann entfernt würden.

Für das Einhängen der Komponente erzeugen wir eine neue JavaScript-Datei, in der wir das »Bootstrapping« unserer Anwendung durchführen.

In diese Datei importieren wir zunächst `React` und `ReactDOM` und natürlich unsere `ChoiceBar`-Komponente.

*ReactDOM.render()*

Die Methode zum Rendern der Komponente heißt `render` und erwartet zwei Parameter: zum einen das native DOM-Element, unterhalb dessen React deine Komponente einfügen soll, und als zweiten Parameter deine Komponente.

Die Komponente können wir wieder direkt in JSX-Syntax schreiben, das entsprechende DOM-Element wählen wir mit der Standard-DOM-Methode `getElementById` aus. In der `index.html`-Datei, die von Create React App generiert wird und die sich direkt im Verzeichnis `public` befindet, ist bereits ein entsprechendes »Einstiegselement« mit der Id `root` dafür angelegt.

Lege nun also bitte die Einstiegsdatei mit dem Namen `index.js` direkt im `src/client`-Verzeichnis an:

```
import React from “react”;
```

```
import ReactDOM from “react-dom”;

import ChoiceBar from “./components/ChoiceBar”;

ReactDOM.render(

  <ChoiceBar />,

  document.getElementById(“root”)

);
```

Wirf nun einen Blick in die Datei `index.html` im `public`-Verzeichnis, die von Create React App erzeugt wurde.

```
<!DOCTYPE html>

<html lang=“en”>

  <head>

    <meta charset=“utf-8” />

    <link rel=“shortcut icon” href=”%PUBLIC_URL%/favicon.ico” />

    <title>React Vote Example</title>

  </head>

  <body>

    <div id=“root”></div>

  </body>
```

```
</html>
```

Wie du siehst, handelt es sich hierbei um eine ganz gewöhnliche HTML-Datei (oben zur besseren Lesbarkeit leicht gekürzt). Beachte aber die drei folgenden Punkte:

- In dem HTML-Dokument gibt es ein (leeres) `div`-Element mit der Id `root`. Das ist genau das `div`, das wir in der `render`-Methode referenziert haben. Unterhalb dieses Elements wird React zur Laufzeit unsere Komponente einhängen (das `div`-Element selbst bleibt bestehen).
- Während des Build-Prozesses fügt Webpack vor das schließende `body`-Tag einen oder mehrere neue `script`-Tags ein, die den kompilierten Code der Anwendung referenzieren. Wie viele Dateien erzeugt und wie diese benannt werden, wird von Webpack erst zur Build-Zeit festgelegt.
- Als letzten Eintrag vor dem schließenden `head`-Element wird von Webpack zur Laufzeit noch der CSS-Code unserer Anwendung eingebunden (siehe unten).

**Anwendung starten**

Wir können nun unsere »Anwendung« übersetzen lassen und den Webserver dafür starten. Dazu führe bitte aus dem Root-Verzeichnis deines Projekts folgendes `npm`-Kommando aus:

```
npm start
```

Sobald der Webserver läuft, kannst du die Komponente im Webbrowser unter der Adresse http://localhost:8080 ansehen[1].

Wenn du in Chrome oder Firefox die React Developer Tools installiert hast, kannst du dir Informationen über die Komponente anzeigen lassen, indem du auf das »Components«-Tab in den Entwicklertools deines Browsers klickst und in dem Komponentenbaum auf der linken Seite die `ChoiceBar`-Komponente auswählst.

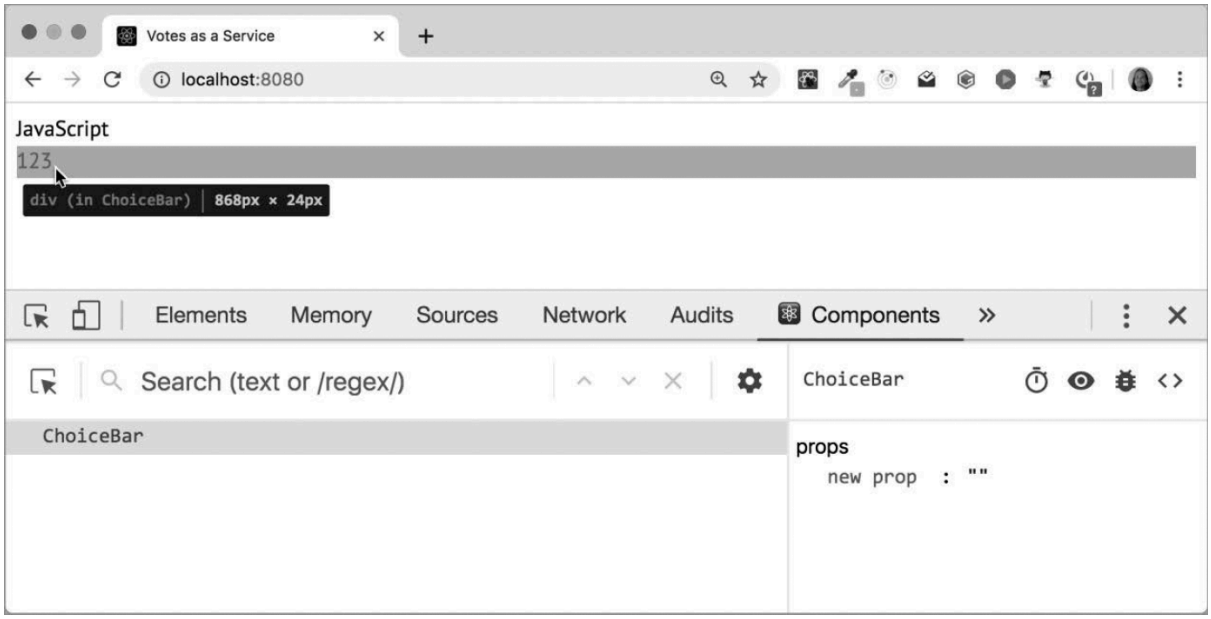


***Abb. 4–2*** *Die ChoiceBar im Chrome-Browser: oben die gerenderte Komponente, unten die React Developer Tools*

### Schritt 4: Styling der Komponente

*CSS-Angaben*

Zum jetzigen Zeitpunkt ist das Aussehen der `ChoiceBar` noch weit von dem entfernt, wie wir uns die Komponente in der Anwendung vorstellen. Wir wollen das jetzt ändern und zeigen dir dazu, wie du in einer React-Komponente CSS-Klassen und -Eigenschaften verwenden musst. Den eigentlichen CSS-Code sehen wir uns nicht an, dieser liegt bereits fertig im `hands-on`-Verzeichnis in der Datei `index.css`.

*CSS-Klassen angeben: className statt class*

Zunächst schreiben wir die benötigten CSS-Klassen an die `div`-Elemente unserer Komponente. Da die Komponente den prozentualen Anteil der Stimmen an der Anzahl der Gesamtstimmen der Umfrage anzeigen soll, müssen wir bei dieser Gelegenheit auch noch ein weiteres `div`-Element hinzufügen, das als »Fortschrittsbalken« (Progressbar) dient. Ändere dazu die Komponentenfunktion wie folgt. Bitte beachte dabei, dass CSS-Klassen in JSX nicht mit dem Attribut `class` (wie von HTML-Elementen gewohnt), sondern mit dem Property `className` definiert werden:

```
export default function ChoiceBar() {

  return <div className=“ChoiceBar”>

    <div className=“Progress”>
```

```
      <div className="ChoiceBarTitle">JavaScript</div>
    </div>
    <div className="ChoiceBarBadge">123</div>
  </div>;
}
```

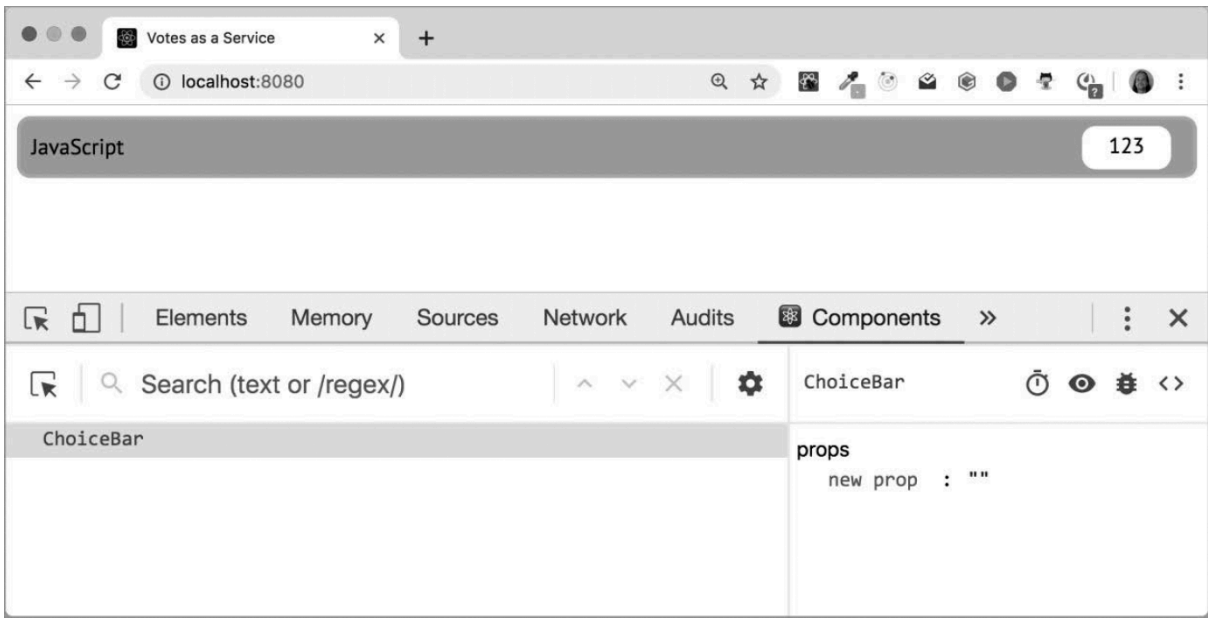


***Abb. 4–3*** *Die ChoiceBar-Komponente mit CSS-Klassen*

Nun fehlt noch die Anzeige für die prozentuale Verteilung der abgegebenen Antworten. Der Balken soll bei 100% der Stimmen ganz von links nach rechts gehen, bei 0% der Stimmen soll gar kein Balken vorhanden sein und bei allen Werten dazwischen soll der Balken entsprechend proportional gefüllt sein. Dieses Verhalten erreichen wir ganz einfach mit der CSS-Eigenschaft `width`.

*CSS-Eigenschaften als Objekt angeben*

In React werden CSS-Eigenschaften mit einem JavaScript-Ausdruck als Objekt angegeben. Der Name der CSS-Eigenschaft wird dabei als Key und der gewünschte Wert der Eigenschaft als Wert im Objekt-Literal verwendet. Für die `width`-Eigenschaft brauchen wir also ein Objekt mit der folgenden Eigenschaft, wenn wir – zunächst hartcodiert – einen Balken mit 37% Breite erzeugen wollen: `{"width": "37%"}`.

Da JavaScript-Ausdrücke in JSX mit geschweiften Klammern umschlossen werden müssen, verwenden wir bei der Angabe des `style`-Attributs zwei geschweifte Klammern hintereinander: die erste zur Einleitung des JavaScript-Codes, die zweite für das Objekt-Literal.

Unsere Funktion sieht damit wie folgt aus:

```
export default function ChoiceBar() {

  return <div className=“ChoiceBar”>

    <div className=“Progress” style={{“width”: “37%”}}>

      <div className=“ChoiceBarTitle”>JavaScript</div>

    </div>

    <div className=“ChoiceBarBadge”>123</div>

  </div>;

}
```

*CSS-Datei importieren*

Damit die CSS-Angaben wirken können, muss die (fertige) CSS-Datei in unsere Anwendung eingefügt werden. Dafür gibt es mehrere Möglichkeiten. Ein gängiger Weg ist es, die entsprechende CSS-Datei im Startcode der Anwendung mittels `import` zu referenzieren. Webpack kann diese `import`-Anweisung auflösen (obwohl sie nicht auf eine JavaScript-, sondern eine CSS-Datei zeigt) und fügt den CSS-Code in unsere `index.html`-Datei ein.

Ergänze aus diesem Grund die `index.js`-Datei um folgende `import`-Anweisung:

```
import React from “react”;

import ReactDOM from “react-dom”;
```

```
import "./index.css";

…
```

Nach dem Speichern dieser Änderung sollte Webpack die Anwendung automatisch neu bauen und die Darstellung im Browser aktualisieren. Dort sieht die `ChoiceBar` nun schon sehr viel besser aus:

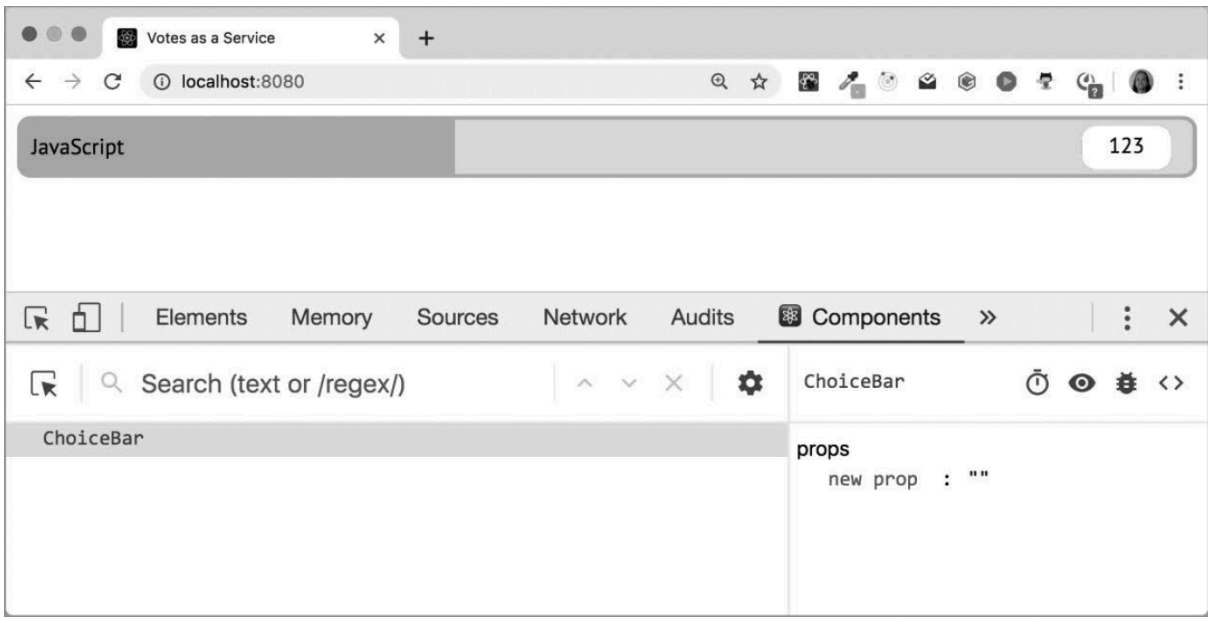


***Abb. 4–4*** *Die »gestylte« ChoiceBar. Der linke Balken gibt den (fiktiven) prozentualen Anteil der 123 Stimmen an der Gesamtzahl aller abgegebenen Stimmen an.*

Wenn du mit den Developer Tools deines Browsers den DOM der Anwendung anschaust, siehst du dort neben der gerenderten Komponente auch das CSS im `head`-Element und die JavaScript-Dateien vor dem `body`-Element, die Webpack erzeugt hat.

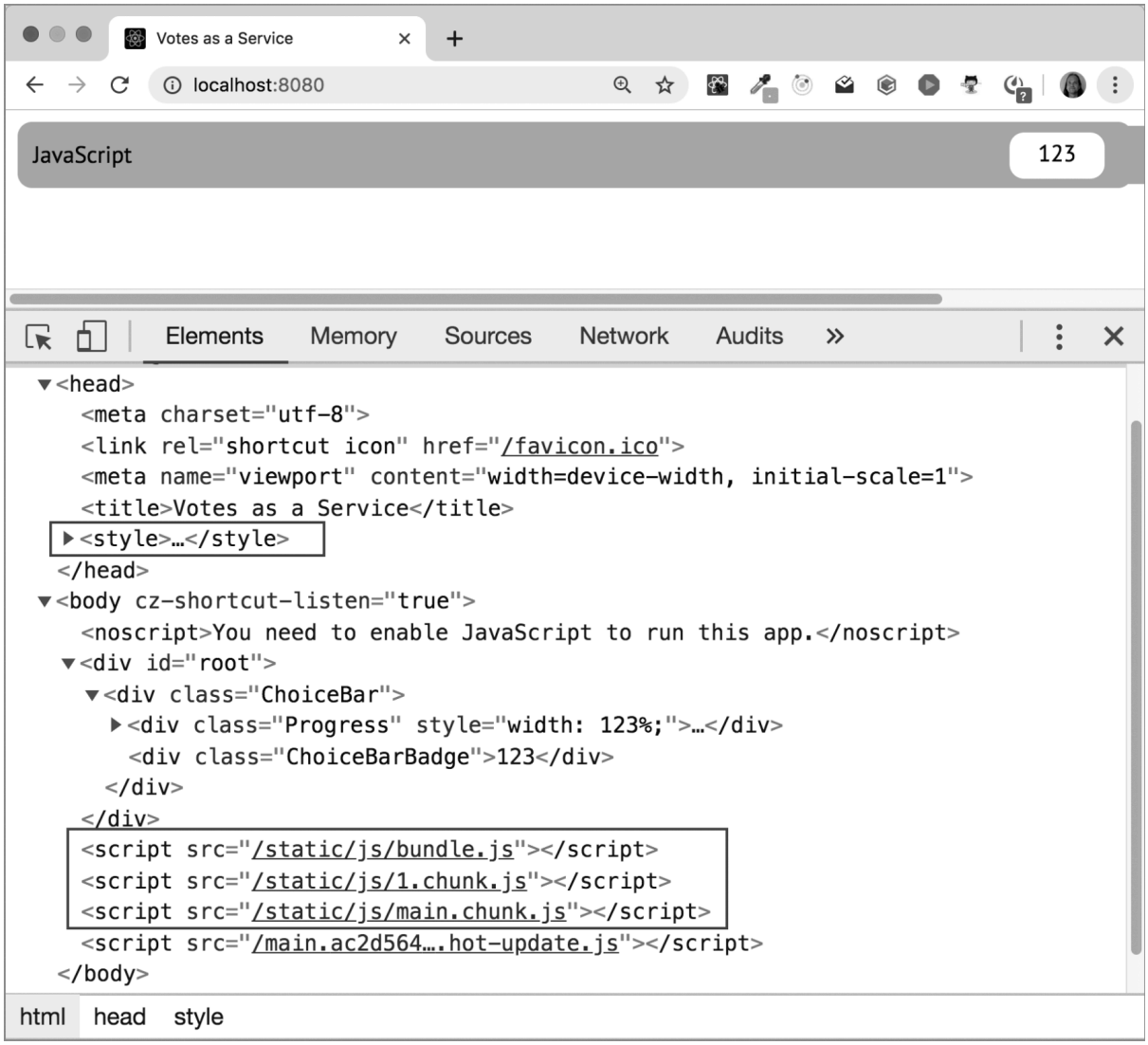


***Abb. 4–5*** *CSS- und JavaScript-Code wurden von Webpack hinzugefügt.*

### Schritt 5: Die Komponente mit Properties konfigurieren

In optischer Hinsicht erfüllt unsere Komponente nun unsere Anforderungen. Allerdings sind die dargestellten Werte (Titel, Anzahl Stimmen, prozentualer Anteil) noch in der Komponente fest eingetragen. Das wollen wir jetzt ändern, denn die Werte sollen der Komponente von außen übergeben werden. Werte, die von außen an die Komponente übergeben werden, werden in React *Properties* genannt.

*Properties*

Die Properties werden einer Komponente als (optionaler) Methodenparameter (üblicherweise `props` genannt) übergeben. Dabei handelt es sich um ein Objekt, das als Schlüssel die Namen der Properties und als Wert den Wert des jeweiligen Propertys enthält.

Die `ChoiceBar`-Komponente benötigt zwei Properties: den anzuzeigenden Titel (`title`) und die Prozentzahl (`percent`). Die Anzahl Stimmen lassen wir zunächst hartcodiert, diese werden wir im nächsten Schritt in den Zustand (State) der Komponente übernehmen.

Die Funktion könnte dazu wie folgt aussehen:

```
export default function ChoiceBar(props) {

  // Übergebene Properties stehen als

  // props.title und props.percent

  // zur Verfügung

}
```

*ES6 Destructuring beschreiben wir im Anhang*

Um allerdings einerseits die Verwendung der Properties in der Funktion zu vereinfachen und andererseits schon in der Signatur explizit auszudrücken, welche Properties die Komponente erwartet, schreiben wir als Methodenparameter nicht `props`, sondern verwenden das seit ES6 mögliche Objekt-Destructuring. Dazu wird in geschweiften Klammern als Methodenparameter angegeben, welche Properties die Funktion aus dem übergebenen `props`-Objekt verwenden soll. Die Werte aus dem Objekt stehen dann unter gleichnamigen Variablen in der Funktion zur Verfügung:

```
export default function ChoiceBar({title, percent}) {

  // Übergebene Properties stehen als

  // title und percent zur Verfügung

}
```

*Auf Properties zugreifen*

Um auf die Properties in der Funktion innerhalb eines JSX-Ausdrucks zuzugreifen, müssen die Namen der

Properties wieder in geschweifte Klammern geschrieben werden, da es sich hierbei um einen JavaScript-Ausdruck handelt. Ohne Destructuring wäre das also `{props.title}` und mit Destructuring lediglich `{title}`.

*Beispiel: Verwenden von Properties*

Um die benötigten Properties für die `ChoiceBar` einzuführen, nimm jetzt bitte zwei Änderungen an unserer Komponente vor: Ändere die Methodensignatur und ersetze dann in der Methode die bislang hartcodierten Werte durch die übergebenen Properties:

```
export default function ChoiceBar({title, percent}) {

  return <div className=“ChoiceBar”>

    <div className=“Progress” style={{‘width’: percent + ‘%’}}>

      <div className=“ChoiceBarTitle”>{title}</div>

    </div>

    <div className=“ChoiceBarBadge”>123</div>

  </div>;

}
```

Damit die Komponente nun ordnungsgemäß aufgerufen wird, ergänze bitte in der Datei `index.js` die von der Komponente geforderten Properties.

Dabei werden die Properties nicht als Objekt, sondern hintereinandergeschrieben (genau wie die Attribute bei HTML-Elementen).

```
…

ReactDOM.render(

  <ChoiceBar title=“JavaScript” percent={37}/>,
```

```
  document.getElementById("root")

);
```

Nach dem Speichern der Änderungen wird die Komponente automatisch neu gerendert.

### Schritt 6: Zustand

Wenn auf den `ChoiceBar`-Balken geklickt wird, soll die Anzahl der für diese Option abgegebenen Stimmen um eins erhöht werden und die neue Anzahl soll entsprechend dargestellt werden. (Natürlich soll dann auch die prozentuale Verteilung in `percent` angepasst werden, aber darum kümmern wir uns im nächsten Kapitel.)

*State*

Daten innerhalb einer Komponente, die von der Komponente selbst verwaltet werden und sich verändern können, werden in React Zustand oder State genannt. In anderen Frameworks wird der Zustand häufig auch Model genannt.

Da eine Funktion (wie unsere Komponentenfunktion auch) normalerweise zustandslos ist, benötigen wir einen Weg, um den Zustand, also die Daten unserer Komponente, zwischen mehreren Aufrufen der Komponente zu erhalten (die Komponentenfunktion wird von React diverse Male erneut aufgerufen, zum Beispiel immer, wenn sich Properties oder State verändern!).

*Hooks*

Die Lösung für dieses Problem ist der `useState`-Hook. *Ein Hook ist in React eine besondere Funktion, mit der du dich in den Lebenszyklus einer Komponente einklinken (*»einhaken«*) kannst*. Im Fall des `useState`-Hooks sorgt React einerseits dafür, dass der aktuelle Zustand auch über mehrere Renderzyklen (also Ausführungen der Komponentenfunktion) hinweg erhalten bleibt. Zum anderen wird uns über den Hook eine Setter-Funktion zur Verfügung gestellt, mit der wir den aktuellen State verändern und React gleichzeitig mitteilen können, dass die Komponente neu gerendert werden muss, damit der neue Wert angezeigt werden kann.

*Beispiel: useSate*

In unserem Beispiel wollen wir die aktuelle Anzahl Stimmen (`count`) im Zustand halten. Passe dazu bitte erneut die `ChoiceBar`-Komponente an: Zum einen musst du mit `useState` einen Zustand erzeugen. Den aktuellen Zustand verwenden wir dann, um ihn in der Komponente anzuzeigen (aktuell steht dort ein hartcodierter Wert). Außerdem soll der Zustand verändert werden, sobald auf die Komponente geklickt wird. Dazu registrieren wir am äußeren `div` der Komponente einen Event Handler

für Click-Ereignisse. Sobald der Handler aufgerufen wird, aktualisieren wird den Zustand, indem wir mit der Setter-Funktion den Zustand um eins erhöhen. Daraufhin rendert React die Komponente neu und wir können den neuen Wert anzeigen.

```
import React from “react”;

export default function ChoiceBar({ title, percent }) {

  // Zustand erzeugen. 0 ist der initiale Wert

  const [count, setCount] = React.useState(0);

  function handleClick() {

    setCount(count + 1);

  }

  return (

    <div className=“ChoiceBar” onClick={handleClick}>

      <div className=“Progress” style={{ width: percent + “%”
      }}>

        <div className=“ChoiceBarTitle”>{title}</div>

      </div>

      <div className=“ChoiceBarBadge”>{count}</div>

    </div>

  );
```

```
}
```

Bitte beachte, dass der Wert, den du `useState` als Parameter übergibst, nur beim ersten Rendern der Komponente verwendet wird. Wenn React die Funktion bei weiteren Render-Vorgängen erneut ausführt, wird dieser Wert verworfen. React merkt sich dafür, welche `useState`-Aufrufe (und Aufrufe anderer Hooks) in einer Funktion ausgeführt wurden, um die entsprechenden Werte beim erneuten Ausführen der Funktion wieder zur Verfügung stellen zu können (innerhalb der Funktion selbst kann React die Daten nicht speichern, da diese über die Dauer der Funktionsausführung hinaus nicht zur Verfügung stehen).

Wenn du die Komponente speicherst und erneut im Browser aufrufst, kannst du den Zähler nun durch Klicken auf die Komponente erhöhen.

***Abb. 4–6*** *Das Beispielprojekt*

Du findest den fertigen Code nach diesem Stand im Repository unter `schritte/04_eine_react_komponente.`

## 4.2 React-Komponenten in der Übersicht

### 4.2.1 Importieren und Verwenden von React

Wenn du React-Komponenten schreibst, musst du zwingend React unter dem Namen `React` importieren, da der kompilierte Code, den Babel oder TypeScript erzeugen, das React-Objekt genau unter diesem Namen erwartet:

```
import React from “react”;
```

Wenn du in deiner Datei weitere Elemente aus dem React-Objekt benötigst, kannst du diese entweder zusätzlich explizit mit einem benannten Import importieren (Variante 1), oder du verwendest sie direkt aus dem React-Objekt (Variante 2). Im folgenden Beispiel wird die `useState`-Funktion importiert sowie die `Suspense`-Komponente.

```
// Variante 1: benannter Import

import React, { useState, Suspense } from “react”;

function MyComponent() {

  // Direkte Verwendung unter dem importierten Namen

  const […] = useState(…)

  return <Suspense>…</Suspense>;

}

// Variante 2: Zugriff über React-Objekt

import React from “react”;

function MyComponent() {

  // Verwendung mit React als “Namespace”

  const […] = React.useState(…)

  return <React.Suspense>…</React.Suspense>;

}
```

Welche der beiden Varianten du wählst, bleibt dir überlassen. Wir werden im Buch die zweite Variante verwenden, damit du in den Codebeispielen auf einen Blick erkennen kannst, dass eine Funktion oder Komponente von React kommt.

## 4.3 Hooks

Mit der React-Version 16.8, die Anfang 2019 veröffentlicht wurde, wurde eine neue API vorgestellt, mit der Funktionskomponenten um nahezu alle Features erweitert wurden, die vorher der Klassen-API vorbehalten waren. Diese API trägt den Namen Hooks, da man sich damit in die Funktionsweise und den Lebenszyklus von Funktionskomponenten einklinken (»einhaken«) kann. Mit Hooks ist es beispielsweise möglich, Zustand in Funktionskomponenten zu halten, obwohl Funktionen per se erst einmal zustandslos sind. Die Hooks-API verspricht außerdem eine bessere Wiederverwendbarkeit von Code als in Klassenkomponenten. Dazu lässt sich mehrfach genutzter Code in selbst geschriebenen Hooks auslagern.

### 4.3.1 React Hooks

React bringt einige Hooks von Haus aus mit und auch einige Bibliotheken aus dem React-Umfeld bieten mittlerweile Hooks an (dazu gehören der React Router, Redux und Apollo GraphQL).

Wichtige React Hooks, die wir uns auch im Laufe des Buchs ansehen werden, sind:

- `useState`: Mit dem `useState`-Hook kannst du in deiner Funktionskomponente Zustand verwalten. Dieses Feature war bis vor React 16.8 den Klassenkomponenten vorbehalten (mehr dazu in Abschnitt 4.3).
- `useEffect`: Hiermit kannst du Seiteneffekte in deiner Komponente implementieren. Seiteneffekte sind Verhalten, die über das reine Rendering einer Komponente hinausgehen. Dazu gehört z.B. das Lesen von Daten von einem Server oder das Arbeiten mit nativen DOM- und Browser-Elementen (mehr dazu in Abschnitt 7.2).
- `useRef`: Dieser Hook erlaubt es dir, allgemeine Referenzen in deiner Komponente zu halten, die z.B. über mehrere Renderings hinweg gültig bleiben (siehe Abschnitt 6.5).
- `useMemo` und `useCallback`: Diese beiden Hooks dienen der Performance-Optimierung. Du kannst damit beispielsweise Werte cachen, die besonders teuer zu berechnen sind und nicht mit jedem Rendering erneut berechnet werden können oder sollen (siehe Abschnitt 5.5).

- `useContext`: Mit `useContext` kannst du einen oder mehrere Kontexte in deiner Komponente verwenden, die ein Provider zur Verfügung gestellt hat (siehe Abschnitt 9.6).
- Die vollständige Liste der eingebauten React Hooks findest du in der Online-Dokumentation[2].

### 4.3.2 Regeln für Hooks

Hooks sind technisch betrachtet normale JavaScript-Funktionen, die du in deiner Komponente je nach Bedarf aufrufen kannst. Es gibt jedoch einige Besonderheiten zu beachten:

- Die Namen von Hooks, also deren Funktionsnamen, fangen immer mit dem Wort `use` an, damit – unter anderem für das Tooling – eine Funktion als Hook erkennbar bleibt.
- Hooks können nur innerhalb von Funktionskomponenten oder anderen Hooks verwendet werden. Du kannst keine Hooks in Klassenkomponenten verwenden.
- Hooks können nur auf oberster Ebene einer Komponentenfunktion verwendet werden. Es ist beispielsweise nicht erlaubt, sie in `if`-Abfragen, Schleifen oder Funktionen, die keine Hooks sind, zu verwenden (dazu unten weitere Beispiele).

Im Folgenden findest du einige Beispiele für die korrekte und inkorrekte Verwendung von Hooks. Auch wenn die Beispiele nur den `useState`-Hook verwenden, gelten die Regeln für alle Hooks, auch für deine eigenen.

```
// ERLAUBT

function VoteEditor(props) {

  const [title, setTitle] = React.useState(””);

  const [description, setDescription] = React.useState(””);

  …

}
```

```
// EBENFALLS ERLAUBT

// (Hook-Aufrufe werden zwar “unterbrochen”, sind aber

// weiterhin auf oberster Ebene)

function VoteEditor(props) {

  const [title, setTitle] = React.useState(””);

  const uppercaseTitle = title.toUpperCase();

  const [description, setDescription] = React.useState(””);

  …

}

// VERBOTEN

// (Hook-Aufrufe dürfen nicht in if-Abfragen, Schleifen etc.
stehen)

function VoteEditor(props) {

  const [title, setTitle] = React.useState(””);

  if (title !== ””) {

    const [description, setDescription] = React.useState(””);

    …

  }
```

```
  …
}
// VERBOTEN
// (Zweiter Hook-Aufruf abhängig von if-Bedingung zuvor)
function VoteEditor(props) {
  const [title, setTitle] = React.useState(“”);
  if (title === “”) {
    return <div>Please enter Title</div>
  }
  const [description, setDescription] = React.useState(“”);
  …
}
// ERLAUBT
// (“useInitVote” ist ein Custom Hook)
function useInitVote(initialValue) {
  return React.useState(initialValue || “”);
}
```

```
function VoteEditor(props) {

  const [title, setTitle] = useInitVote();

  …

}

// VERBOTEN

// (“initVote” könnte ein Custom Hook sein, müsste dann aber

// “useInitVote” heißen)

function initVote(initialValue) {

  return React.useState(initialValue || ””);

}

function VoteEditor(props) {

  const [title, setTitle] = initVote();

  …

}
```

Um fehlerhafte Verwendung von Hooks zu vermeiden, solltest du das ESLint-Plug-in eslint-plugin-react-hooks[3] verwenden. Wenn du dein Projekt mit Create React App erzeugst, ist dieses Projekt bereits bei dir installiert.

## 4.4 Zustand verwalten mit useState

Im Hands-on-Teil haben wir bereits mit dem Zustand, in React State genannt, unserer Komponenten gearbeitet und einen Counter in der `ChoiceBar` implementiert. Andere Beispiele für den Zustand einer Komponente sind zum Beispiel der aktuelle Wert eines Eingabefelds, die Information, ob eine Checkbox angehakt ist, oder auch Daten, die von einem Server gelesen wurden.

Werte und Daten, die aus anderen Daten abgeleitet werden können, sollten allerdings nicht in den State, um Redundanzen und damit mögliche Inkonsistenzen zu vermeiden. Die Information, ob ein Button enabled ist oder nicht, kann in der Regel anhand der Daten im State abgeleitet werden (zum Beispiel, weil ein Wert leer oder `null` ist) und sollte nicht im State hinterlegt werden. Ansonsten könnte es passieren, dass der zugrunde liegende Wert verändert wird, aber vergessen wird, die enable-Information ebenfalls anzupassen. Wenn diese Information hingegen unmittelbar beim Rendern aus dem State abgeleitet wird, kann das nicht passieren.

#### Der useState-Hook

Um mit State in deiner Funktionskomponente zu arbeiten, verwendest du den `useState`-Hook:

```
import React from “react”;

function MeineKomponente(…) {

  const [ currentValue, setValue ] =
  React.useState(initialValue);

  …

}
```

Dieser Hook liefert zwei Parameter in einem Array zurück:

- Der erste Eintrag entspricht dem aktuellen Wert des Zustands. Wird die Komponente das erste Mal gerendert, wird dieser Wert auf den initialen Wert gesetzt, den du dem `useState`-Hook als Parameter übergibst.

- Der zweite Eintrag in dem Array ist eine Setter-Funktion, mit der du den Zustand verändern kannst. Die Ausführung der Funktion und das damit einhergehende Verändern des Zustands führt dazu, dass deine Komponente neu gerendert wird, die Komponentenfunktion wird also erneut aufgerufen.

Was du für einen Wert bzw. für einen Typ (String, Number, Array etc.) in einem Zustand ablegst, bleibt dir überlassen. Für komplexere Komponenten kannst du auch mehrere Zustände mit `useState` erzeugen und individuell verändern. Ein Beispiel dazu sehen wir in Kapitel 6.

### Lebenszyklus einer Komponente

Sehen wir uns noch einmal die `ChoiceBar`-Komponente in ihrer derzeitigen Version an, um den Lebenszyklus der Komponente zu untersuchen:

```
export default function ChoiceBar({ title, percent }) {

  const [count, setCount] = React.useState(0);

  function handleClick() {

    setCount(count + 1);

  }

  return (

    <div className=“ChoiceBar” onClick={handleClick}>

      <div className=“Progress” style={{ width: percent + “%”
      }}>

        <div className=“ChoiceBarTitle”>{title}</div>

      </div>

      <div className=“ChoiceBarBadge”>{count}</div>
```

```
    </div>

  );

}
```

Der Lebenszyklus dieser Komponente sieht wie folgt aus:

1. Die Komponente soll erstmalig gerendert werden. Das kann ausgelöst werden, weil sie direkt mit `ReactDOM.render` in den DOM gehängt wird oder weil die Komponente von einer anderen (Parent-)Komponente verwendet und gerendert wird.
2. Die Komponentenfunktion wird von React ausgeführt und `useState` wird aufgerufen. Da es sich um den *ersten* Renderzyklus handelt, wird der mit dem Paramter übergebene initiale Werte (`0`) als aktueller Wert (`count`) in die Komponente zurückgeliefert.
3. Die Funktion gibt mittels JSX die aktuelle UI zurück. In dieser wird der aktuelle `count`-Zustand angezeigt. React aktualisiert den nativen DOM im Browser derart, dass die UI angezeigt wird.
4. Sobald nun auf das entsprechende `div` geklickt wird, wird die von `useState` zurückgelieferte Setter-Funktion (`setCount`) verwendet, um den Zustand der Komponente zu aktualisieren (erhöhen von `count` um 1).
5. Der Aufruf von `setCount` veranlasst React, die Komponente neu zu rendern, was technisch bedeutet, dass die Funktion (`ChoiceBar`) erneut aufgerufen wird.
6. Der `useState`-Aufruf liefert nun in diesem *zweiten* Renderzyklus nicht mehr den initialen Wert zurück (wie noch in Schritt 2), sondern den in Schritt 4 gesetzten neuen Werten (also `1`).
7. Die Komponente liefert die aktualisierte UI zurück, woraufhin React den DOM entsprechend anpasst – die Aktualisierungen, insbesondere der erhöhte Counter, ist im Browser nun sichtbar.
8. Diese »Schleife« (Schritte 4 bis 7) wiederholt sich so lange, bis die Komponente nicht mehr angezeigt wird.
   (Unsere Komponente bleibt derzeit »ewig« im DOM, da es in der »Anwendung« noch keine Möglichkeit gibt, eine andere Komponente anzuzeigen bzw. die `ChoiceBar` auszublenden.)

**Achtung: »Rendern« hat doppelte Bedeutung!**

Das Wort »rendern« bzw. »Rendering« hat in React eine doppelte Bedeutung: Zum einen ist damit gemeint, dass React eine Komponentenfunktion (bzw. die render-Methode in einer Komponentenklasse) aufruft, die eine virtuelle DOM-Struktur zurückgibt – in der Regel durch die Verwendung von JSX.

Zum anderen ist das »native« Rendering gemeint, also die Manipulation des echten DOM im Browser, damit dieser die Komponente auch darstellt.

Sofern nicht anders angegeben, bezieht wir uns beim Wort »rendern« auf die *erste* Bedeutung.

### Initialisieren des Zustands

In den gezeigten Beispielen übergeben wir dem `useState`-Hook als Parameter den initialen Wert deines Zustands. Wie wir aber gesehen haben, wird der Wert nur beim ersten Rendern der Komponente überhaupt benötigt, in allen folgenden Renderzyklen wird er verworfen. Das ist bei einfachen initialen Werten auch kein Problem. Ist der Wert allerdings sehr aufwendig zu erzeugen, kannst du dem `useState`-Hook als Parameter auch eine Callback-Funktion übergeben. Diese wird *nur* beim *ersten* Rendern ausgeführt und muss dann den initialen Wert erzeugen und zurückliefern.

```
function MeineKomponente(props) {

  function getInitialValue() {

    // teure Ermittlung des initialen Wertes hier

    const initialValue = …;

    return initialValue;

  }

  const [ value, setValue ] = React.useState(getInitialValue);

}
```

Diese Variante solltest du – wie alle Optimierungen – nur dann wählen, wenn du an dieser Stelle Probleme mit der Performance durch das ständige Neurendern hast.

### Aktualisieren des Zustands

Der Setter-Funktion, die `React.useState` zurückliefert, kannst du übrigens nicht nur einen neuen Wert übergeben, sondern auch eine Callback-Funktion. Diese Callback-Funktion wird dann von React aufgerufen und der aktuelle Wert deines Zustands wird der Funktion als Parameter übergeben. Die Callback-Funktion liefert dann den neuen Zustand zurück. Dieses Verhalten kannst du für spezielle Fälle nutzen, etwa wenn du mit asynchronem Code arbeitest (mehr dazu in Kapitel 7). Auf der anderen Seite kannst du damit aber auch die »fachliche« Verarbeitung des Zustands aus deinen Komponenten herausziehen, um sie besser testen und außerhalb von React wiederverwenden zu können.

Das kannst du dir in dem extrem vereinfachten, folgenden Beispiel ansehen:

```
import React from “react”;

// incrementChoice ist eine “reguläre” JavaScript-Funktion ohne

// Abhängigkeit auf React.

// Du kannst sie z.B. sehr einfach mit einem Unit Test testen

// currentCount ist der aktuelle Wert im Zustand, den React

// beim Aufruf übergibt

function incrementChoice( currentCount => currentCount + 1);

export default function ChoiceBar(…) {

  const [ count, setCount ] = React.useState(0);

  function handleClick() { setCount(incrementChoice); }

  return (

    <div onClick={handleClick}>…</div>
```

```
  );

}
```

## 4.5 Ereignisse und Event Handler

*Beispiel: ein Event Handler*

In React kannst du auf Ereignisse reagieren, indem du in einer Komponente Event Handler für bestimmte Ereignisse registrierst[4]. Das funktioniert ähnlich wie in HTML: Du übergibst den Event Handler als Property an das gewünschte HTML-Element. Im Gegensatz zu HTML werden die Ereignisnamen in React in CamelCase-Notation geschrieben:

```
function Title({title}) {

  return <h1 onClick={ e => console.log('Event: ' + e) }>

    {title}

  </h1>;

}
```

*React Events*

Um auch an dieser Stelle unabhängig vom nativen DOM zu bleiben, sind die Event-Objekte, die von React an deinen Event Handler übergeben werden, keine nativen DOM-Events. Es handelt sich dabei um React-eigene Events, deren Schnittstelle aber identisch mit der Schnittstelle der nativen Elemente ist. Du kannst darauf also alle bekannten Methoden aufrufen und Properties abfragen, wie auch bei den nativen Events. Falls du das native DOM-Event trotzdem einmal benötigst, kannst du es über das Property `nativeEvent` abfragen.

### Hintergrund: globaler Event Listener

*Event-Objekt nicht cachen*

Wenn du einen Event Handler anmeldest, wird dafür von React kein Event Handler im DOM erzeugt. Stattdessen registriert React aus Performance-Gründen genau *einen* globalen Event Handler am obersten Element im (nativen) DOM. Dieser globale Event

Handler verteilt die eingetretenen Events an die registrierten (»fachlichen«) Handler, die in den Komponenten registriert wurden.

In einem Event Handler wird dann üblicherweise der Zustand einer Komponente verändert, indem dort zum Beispiel eine Setter-Funktion eines `useState`-Hooks aufgerufen wird. Der Aufruf wiederum hat zur Folge, dass die Komponente erneut gerendert wird. Der Ablauf sieht dann, beginnend mit dem Auftreten eines nativen Events und endend mit der Aktualisierung des nativen DOM, wie in Abb. 4–7 beschrieben aus. Nun wird wieder gewartet, bis ein Event auftritt, und der Ablauf beginnt erneut.

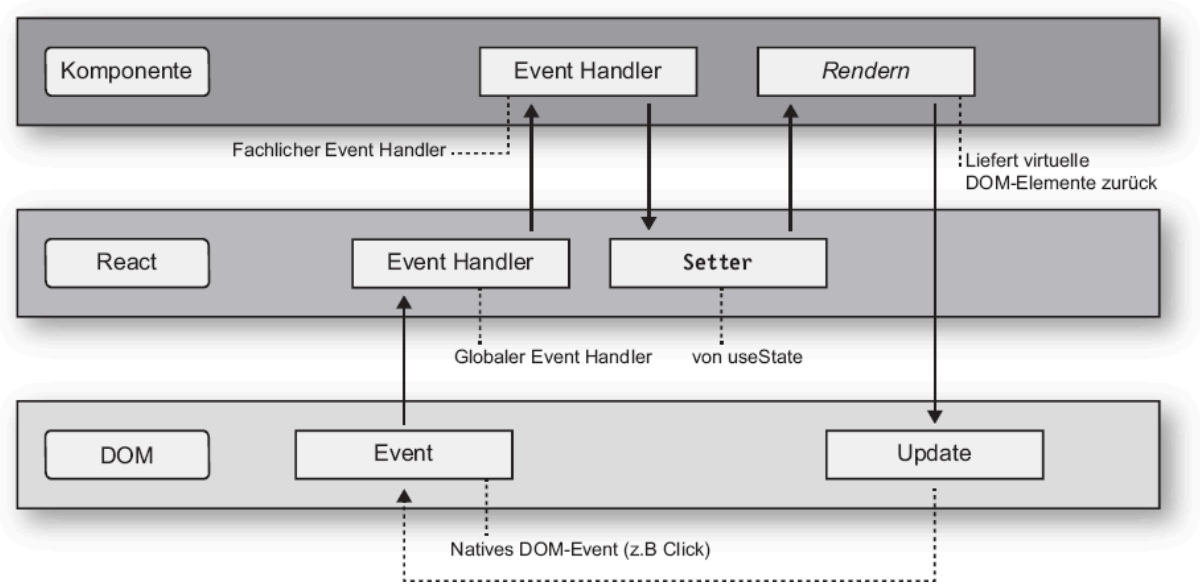


***Abb. 4–7*** *Event-Verarbeitung in React*

## 4.6 JSX zur Beschreibung der UI

*JSX statt Template-Sprache*

React kommt – im Gegensatz zu vielen anderen Webframeworks – ohne Template-Sprache aus. Der UI-Teil einer Komponente, also in erster Linie die HTML-Elemente, aus denen die Komponente besteht, wird stattdessen in JavaScript geschrieben. Um das Schreiben von Elementen zu erleichtern, bringt React eine JavaScript-Spracherweiterung mit dem Namen JSX mit. Mittels dieser Erweiterung lassen sich HTML-Elemente direkt in den JavaScript-Code schreiben. JSX wird mit einem Compiler in gültiges JavaScript übersetzt, sodass der Code in allen Browsern lauffähig ist.

### 4.6.1 Grundlagen

Sehen wir uns zunächst einen sehr einfachen Ausschnitt aus einer fiktiven HTML-Seite an:

```
<h1>Hello, World</h1>
```

Mit JSX kannst du diesen Ausdruck genauso in deinen JavaScript-Code schreiben und zum Beispiel einer Variablen zuweisen:

```
const helloWorld = <h1>Hello, World</h1>;
```

Damit der Code im Browser funktioniert, muss er zunächst von einem Compiler in gültiges JavaScript übersetzt werden. Das daraus erzeugte JavaScript könnte so aussehen:

```
var helloWorld = React.createElement(
    “h1”,
    null,
    “Hello, World”
  );
```

Um die Funktionsweise von JSX zu verstehen, kann es hilfreich sein, wenn du dir in Erinnerung rufst, wie dieser kompilierte Code aussieht. Dann wird klar, dass es sich dabei um reguläres JavaScript handelt, das du überall dort verwenden kannst, wo du einen Funktionsaufruf schreiben kannst.

Um zu sehen, wie dein eigener JSX-Code in kompilierter Form aussieht, kannst du den JSX-Code testweise in den Online-»Playground« des Babel-Compilers[5] oder des TypeScript-Compilers[6] einfügen:

**Compiler**

Um dir den Compiler zu sparen, könntest du diesen Code natürlich auch anstelle des JSX-Codes selber schreiben. Allerdings ist zum einen der JSX-basierte Code sehr viel schlanker und dadurch besser zu lesen und zu schreiben. Zum anderen wird bei der Entwicklung mit React in der Regel ohnehin ein Compiler benötigt, um die von React verwendeten modernen JavaScript-Features in ES5-Code zu übersetzen, die noch nicht von allen Browsern implementiert sind. Auch für diese (experimentellen) Features wird ein Compiler benötigt. Wenn du TypeScript in deinem Projekt einsetzt, kann der TypeScript-Compiler auch den JSX-Code in JavaScript-Code übersetzen.

**Hintergrund: virtueller Dom**

Wie du an dem Beispiel oben gesehen hast, wird aus dem JSX-Code der Aufruf einer `React.createElement`-Methode generiert. Diese Methode liefert allerdings kein `HTMLElement` aus der »echten« DOM-API zurück, wie es beispielsweise die Methode `Document.createElement` in einem HTML-Dokument tun würde. Stattdessen wird eine Instanz von `ReactElement` zurückgeliefert. Dabei handelt es sich um ein sehr einfaches und leichtgewichtiges, unveränderliches JavaScript-Objekt, ein *virtuelles* DOM-Objekt. Dieses virtuelle Objekt wird erst beim *Rendern* einer Komponente in ein konkretes Ausgabeformat – zum Beispiel DOM-Elemente – umgewandelt.

*Trennung von virtuellem und nativem DOM*

Die Trennung von virtuellem und nativem DOM bringt eine Reihe von Vorteilen mit sich. Zunächst einmal kann React dank der leichtgewichtigen React-Elemente sehr schnell feststellen, wo sich innerhalb der Baumstruktur zwischen zwei Render-Phasen überhaupt etwas geändert hat. Es kann dann (prinzipiell teure) Manipulationen im nativen DOM auf ein Minimum beschränken. Nur so ist es überhaupt möglich, dass React es sich »erlauben« kann, bei Änderungen immer ganze Komponenten bzw. Komponentenbäume neu zu rendern.

Außerdem ist es möglich, die React-Elemente in weitere Ausgabeformate neben dem nativen DOM ausgeben zu lassen. So kann React beispielsweise auch HTML-Code in einen String rendern. Mit React Native können die Elemente auch als native Widgets von iOS oder Android gerendert werden.

### 4.6.2 JavaScript-Ausdrücke in JSX

Das oben gezeigte `h1`-Element enthält keinerlei dynamische Informationen. Wenn du in JSX Elemente mit einem variablen Anteil schreiben willst, verwendest du dazu normalen JavaScript-Code. Um zu kennzeichnen, dass du einen JavaScript-Ausdruck verwendest, schreibst du diesen in einem Paar geschweifter Klammern, damit »brichst« du quasi aus JSX wieder aus und bist wieder in JavaScript. In den geschweiften Klammern kannst du gültige JavaScript-*Ausdrücke* (*expressions*) verwenden, aber keine JavaScript-*Anweisungen* (*statements*).

Um z.B. den Text innerhalb des `h1`-Elements aus einer Variablen zu lesen, schreibst du das Element wie folgt:

```
const title = “Hello, World”;

const helloWorld = <h1>{title}</h1>;
```

Natürlich kannst du an der Stelle auch eine Funktion aufrufen:

```
function getTitle() { return “Hello, World”; }

const helloWorld = <h1>{getTitle()}</h1>;
```

Das Gleiche funktioniert auch für Attribute eines Elements. Wenn du den Text rot ausgeben möchtest, gehst du wie folgt vor:

```
const red = “#ff0000”;

const helloWorld = <h1 color={red}>Hello, World</h1>;
```

**Vorsicht bei *class* und *for***

Die beiden in HTML verwendeten Attribute `class` und `for` können nicht direkt in JSX verwendet werden, da JSX (erweitertes) JavaScript ist und `class` und `for` auch Schlüsselwörter in JavaScript sind. Stattdessen müssen dafür die Attribute `className` bzw. `htmlFor` an einem Element verwendet werden:

```
<label className=“input-hint”

       htmlFor=“password-input”>Passwort:</label>
```

Werden dennoch `class` oder `for` verwendet, gibt React eine entsprechende Warnung auf der Konsole aus.

Auf diese Weise kannst du auch ein anderes JSX-Element einbinden:

```
const helloWorld = <h1>Hello, World</h1>;

const head = <div>{ helloWorld }</div>;
```

### 4.6.3 Kommentare in JSX

Auch Kommentare werden in JSX in einem JavaScript-Ausdruck notiert. Darin kannst du wie von JavaScript gewohnt einen Kommentar schreiben:

```
const helloWorld = <h1>{ /* Hello */ }World</h1>;
```

Innerhalb eines JSX-Elements kannst du die beiden von JavaScript bekannten Notationen für Kommentare verwenden:

```
const helloWorld = <h1 /* Anmerkung */>World</h1>;
```

Kommentare mit // dürfen natürlich nur am Zeilenende vorkommen:

```
const helloWorld = <h1 // Anmerkung

   >World</h1>;
```

### 4.6.4 Fallunterscheidungen

Oftmals ist es erforderlich, Attribute oder Elemente abhängig von Bedingungen zu setzen. Dazu kann der ternäre Operator verwendet werden. In dem folgenden Beispiel wird die CSS-Klasse des erzeugten `div`-Elements abhängig von dem Ergebnis der Funktion `isError` entweder auf `error-message` oder `success-message` gesetzt:

```
function isError() { return … ; }

const message = "…";

const label = <div className={isError()?"error-
message":"success-

message"}>{errorMessage}</div>;
```

Das funktioniert nicht nur in Attributen, sondern auch mit Elementen. Im nächsten Beispiel wird innerhalb des `div`-Elements ein `h1`-Element erzeugt, wenn die Funktion `isTitle true` zurückliefert. Ansonsten wird ein `p`-Element hinzugefügt:

```
function isTitle() { return …; }
```

```
const message = “…”;

const header =

  <div>{isTitle() ? <h1>{message}</h1> : <p>{messsage}</p>}
  </div>;
```

Um abhängig von einer Bedingung zu entscheiden, ob ein Element überhaupt eingebunden werden soll oder nicht, kann `null`, `true` oder `false` verwendet werden. Mit dem folgenden Code wird das `h1`-Element nur hinzugefügt, wenn `isTitle true` zurückliefert. Ansonsten wird dem `div`-Element kein Element als Kind hinzugefügt:

```
function isTitle() { return …; }

const message = “…”;

const header = <div>{isTitle() ? <h1>{message}</h1> : null}
</div>;
```

### 4.6.5 Listen

Ein häufig wiederkehrender Anwendungsfall bei der Entwicklung von Komponenten ist das Erzeugen von Listen. Hierfür kann in JSX die JavaScript-Methode `map` verwendet werden, die für Arrays definiert ist. Sie erwartet eine Callback-Funktion, die für jedes Element aus dem Array aufgerufen wird. Das jeweilige Element wird der Callback-Funktion als Parameter übergeben und die Funktion liefert für dieses Element ein neues Element zurück.

Mit JSX kann die Funktion aber auch genutzt werden, um React-Elemente zu erzeugen und zurückzugeben. So lässt sich daraus beispielsweise eine ungeordnete Liste (`ul`), deren Einträge aus `li`-Elementen bestehen, erzeugen:

```
const items = [“a”, “b”, “c”].map( e => <li>{e}</li> );

const list = <ul>{items}</ul>;
```

**Das key-Attribut**

Bei der Verwendung von Listen gibt es in React noch eine Besonderheit zu beachten: Die einzelnen Elemente aus der Liste müssen eindeutig gekennzeichnet werden, damit React die Verbindung zwischen zwei Instanzen des virtuellen DOM herstellen kann. Dazu wird das Attribut `key` verwendet. Mit diesem Attribut muss jedem Element in der Liste ein Wert zugewiesen werden, der innerhalb der Liste eindeutig ist. In dem eben gezeigten Beispiel kann das das Element selbst sein:

```
const items = [“a”, “b”, “c”].map( (e) => <li key={e}>{e}</li>
);

const list = <ul>{items}</ul>;
```

*Eindeutigkeit des key-Attributs*

Das `key`-Attribut sollte übrigens nicht aus dem Index des Elements in dem Array abgeleitet werden. Wenn der Key dem Index entspricht und die Liste beispielsweise umsortiert wird oder ein Element aus dem Array entfernt oder hinzugefügt wird, würde sich der Key verändern, obwohl das eigentliche Element (und damit auch das resultierende gerenderte Element) unverändert geblieben ist. Das führt dann zu unnötigen Manipulationen im nativen DOM und wirkt sich negativ auf die Performance aus.

Wenn für ein Element in einem Array kein `key`-Attribut angegeben ist, gibt React eine entsprechende Warnung auf der Kommandozeile aus: *Warning: Each child in an array or iterator should have a unique »key« prop. Check the render method of `…`.*

### 4.6.6 Style-Angaben in JSX

Angaben zum Aussehen eines konkreten Elements können in HTML direkt an dem Element vorgenommen werden. Dazu wird dem `style`-Attribute ein String mit den gewünschten CSS-Eigenschaften übergeben:

```
<h1 style=“font-size: 12px; color: red”>Hello, World!</h1>
```

*Style-Angaben als Objekt*

Auch in JSX wird das `style`-Attribut verwendet, um CSS-Eigenschaften an einem Element zu setzen. Dabei gibt es allerdings einige gravierende Unterschiede zum `style`-Attribut an einem HTML-Element.

*Der Typ des* `style`*-Attributs in JSX ist ein Objekt und kein String*. Bei der Angabe entfallen daher die Anführungszeichen. Stattdessen müssen zwei Paar geschweifte Klammern verwendet werden: Mit dem ersten Paar wird – wie oben schon gezeigt – ein JavaScript-Ausdruck in JSX gekennzeichnet. Das zweite Paar ist dann die in JavaScript gewohnte Notation eines Objekt-Literals. Als Schlüssel wird dabei die CSS-Eigenschaft verwendet. Der zugehörige Wert wird wie aus CSS gewohnt angegeben, es muss aber darauf geachtet werden, dass dieser einem gültigen JavaScript-Ausdruck entspricht. Das bedeutet insbesondere, dass String-Angaben mit Anführungszeichen umschlossen werden.

*CamelCase-Notation für CSS-Eigenschaften*

*Die Namen der gesetzten CSS-Eigenschaften müssen in Camel-Case-Notation angegeben werden*. Dabei wird der erste Buchstabe in Kleinbuchstaben angegeben und alle Bindestriche werden entfernt. Buchstaben, die auf einen Bindestrich folgen, werden als Großbuchstabe notiert. Aus `font-size` wird dementsprechend `fontSize`.

*Vendor-Präfixe*

Vendor-Präfixe hingegen werden (bis auf das Vendor-Präfix `ms`) immer mit einem führenden Großbuchstaben notiert. Für die Eigenschaft `webkit-animation` wird dementsprechend der Key `WebkitAnimation` verwendet, für `ms-animation` hingegen `msAnimation`.

**Beispiele für Style-Angaben in JSX**

Hier nun einige Beispiele für die korrekte Verwendung des `style`-Attributs.

Um z.B. die Farbe eines Elements zu setzen, verwenden wir folgenden Ausdruck:

```
const helloWorld = <h1 style={{color: “red”}}>Hello, World</h1>;
```

Um auch die Schriftgröße zu setzen, wird die CSS-Eigenschaft `font-size` verwendet, die in CamelCase-Notation angegeben wird:

```
const helloWorld =

  <h1 style={{color: “red”, fontSize: “12px”}}>Hello,
  World</h1>;
```

Da es sich bei der Style-Angabe um ein reguläres JavaScript-Objekt-Literal handelt, können bei der Angabe der Werte auch Variablen verwendet werden:

```
const titleColor = “red”;

const titleFontSize = 12;

const helloWorld = <h1 style={{

   color: titleColor,

   fontSize: titleFontSize + “px”

  }}>Hello, World</h1>;
```

Und natürlich lässt sich auch das Style-Objekt selbst in einer Variablen ablegen und in der `style`-Attribute-Angabe referenzieren:

```
const titleColor = “red “;

const titleFontSize = 12;

const titleStyle = {

   color: titleColor,

   fontSize: titleFontSize + “px”

};

const helloWorld = <h1 style={titleStyle}>Hello, World</h1>;
```

## 4.7 Rückgabewerte von Komponenten

Eine Komponentenfunktion darf folgende Werte zurückliefern:

Genau ein (!) Root-Element. Das Root-Element muss entweder ein React-Wrapper für ein natives HTML-Element sein (z.B. `<div>`, `<h1>` oder `<input>`) oder eine andere React-Komponente, die jeweils natürlich Kind-Elemente enthalten dürfen:

```
export default function PasswordForm() {

  return (

    <form …>

      <label htmlFor=‘usernameInput’>Your name:</label>

      <input type=‘text’ />

      <label htmlFor=‘passwordInput’>Your password:</label>

      <input type=‘password’ name=‘passwordInput’ />

    </form>

  );

}
```

Erlaubt ist es außerdem, einen String oder einen numerischen Wert zurückzugeben, für die React eine Text-Node im DOM erzeugt.

```
function Hello() {

  return “Hello World”;

}
```

Um zu verhindern, dass ein Element im DOM erzeugt wird, kann entweder `null`, `false` oder `true` zurückgegeben werden. Wenn einer dieser Werte zurückgegeben wird, wird von React kein Element im DOM erzeugt. Die folgende Komponente rendert nichts, sofern keine Message übergeben wurde:

```
function ErrorMessage({message}) {

  return message ? <h1>{message}</h1> : null;

}

// Alternativ:

function ErrorMessage({message}) {

  return !!message && <h1>{message}</h1>;

}
```

Eine Liste von Elementen: Komponenten dürfen auch ein Array von Elementen zurückliefern. Dann muss allerdings wie bereits oben beschrieben jedes dieser Elemente über ein eindeutiges `key`-Attribut verfügen.

In dem folgenden Beispiel wird ein Array mit `li`-Elementen zurückgegeben. Für jeden Eintrag im übergebenen `choices`-Property (ein Array, bestehend aus einem Objekt mit `id`- und `label`-Eigenschaft) wird ein Eintrag in dem Array erzeugt.

```
function ChoiceList({choices}) {

  return choices.map(

    choice => <li key={choice.id}>{choice.label} />

  );

}
```

### Fragmente

Wenn eine Komponente mehr als ein Root-Element zurückgeben muss, aber dafür kein Array verwendet werden kann oder soll, kann als Wrapper ein `React.Fragment`-Element verwendet werden. Dieses fungiert als eine Art »virtuelles« Root-Element, für das selbst kein natives Element in den DOM gerendert wird. Lediglich die Kind-Elemente werden in den DOM gerendert:

Auf diese Weise könnte eine Komponente zum Beispiel ein `h1`- und `h2`-Element zurückliefern:

```
function Header({title, subtitle}) {

  return (

    <React.Fragment>

      <div key=“title” className=“title”>title</div>

      <div key=“subtitle” className=“subtitle”>subtitle</div>

    </React.Fragment>

  );

}
```

Als Kurzschreibweise für `React.Fragment` kann auch ein leeres Paar spitzer Klammern verwendet werden. Dadurch wird der Code etwas kürzer und besser lesbar:

```
function Header({title, subtitle}) {

  return (

    <>
```

```
      <div key=“title” className=“title”>title</div>
      <div key=“subtitle” className=“subtitle”>subtitle</div>
    </>
  );
}
```

### Portale

Eine Komponente kann außerdem ein sogenanntes Portal zurückliefern. Dabei handelt es sich um eine React-Komponente, die außerhalb der umschließenden Komponente an einer ganz anderen frei definierbaren Stelle im DOM gerendert wird. Mit Portalen lassen sich z.B. modale Dialoge oder andere Overlays erzeugen, die in der Regel ganz an den Anfang des DOM eingefügt werden, um sicherzustellen, dass sie »über« allen anderen Elementen liegen.

```
function ModalDialog(props) {
  return ReactDOM.createPortal(
    <div className=‘ModalDialog’>
      <h1>{props.dlgTitle}</h1>
      <button>Close</button>
    </div>),
    document.getElementById(“modalDialogMountPoint”)
      // modalDialogMountPoint muss ein existierendes
```

```
      // Element im DOM sein!
  );
}
```

## 4.8 Einhängen der Anwendung in den DOM

*Beispiel*

Um ein virtuelles React-Element, also eine Repräsentation eines HTML-Elements oder einer eigenen Komponente, in den echten DOM einzuhängen, wird die Methode `ReactDOM.render` verwendet. Diese Methode erwartet zunächst das `ReactElement`, das in den DOM eingehängt werden soll, und als zweiten Parameter einen existierenden Knoten aus dem nativen DOM, unterhalb dessen React das Element rendern soll. In dem folgenden Beispiel muss im nativen DOM ein Element mit der Id `titleElement` vorhanden sein. Unterhalb dieses Elements wird React die `Header`-Komponente rendern:

```
import React from “react”;
import ReactDOM from “react-dom”;
function Title(props) {
  return <h1>{props.label}</h1>;
}
function Header() {
  return <div className=“header “>
    <Title label=“Hello, World” />
  </div>;
```

```
}

ReactDOM.render(

  <Header />,

  document.getElementById(“titleElement”)

);
```

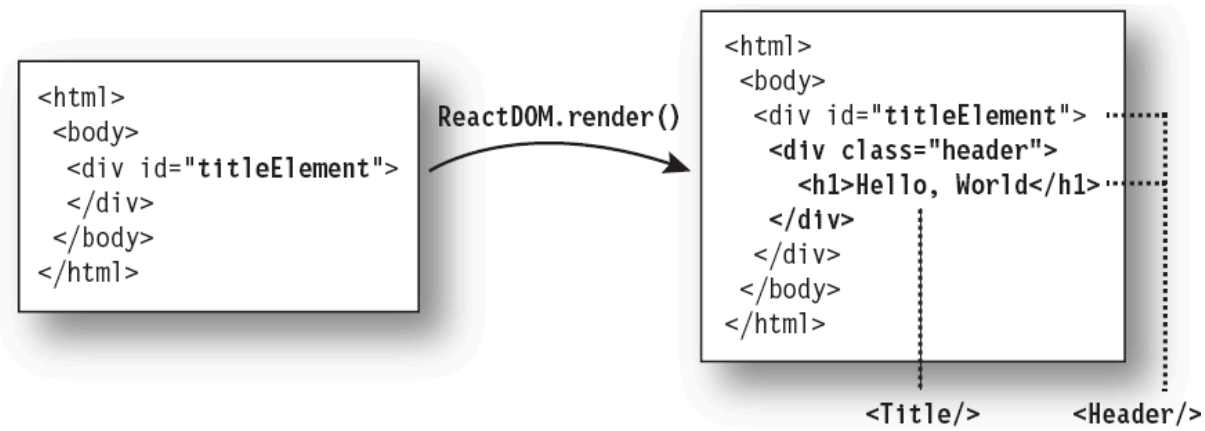


***Abb. 4–8*** *ReactDOM.render fügt die gerenderte Komponente in den DOM ein.*

Wenn die `render`-Funktion das erste Mal aufgerufen wird, ersetzt React alle eventuell bestehenden DOM-Elemente unterhalb des ausgewählten Elements im nativen DOM und fügt stattdessen die Elemente ein, die sich aus dem virtuellen DOM der Komponente ergeben.

Wird die `render`-Funktion mit der gleichen Komponente und dem gleichen Zielelement nochmals aufgerufen, vergleicht React die beiden übergebenen React-Elemente (also das Element aus dem vorherigen Aufruf und das Element, das beim jetzigen Aufruf als erster Parameter an die `render`-Methode übergeben wird) und ändert im nativen DOM nur die DOM-Elemente, deren virtuelle Repräsentation sich verändert hat.

Wenn du also genau denselben Aufruf wie oben gezeigt noch mal durchführen würdest, fänden keine Manipulationen des nativen DOM statt. Der DOM bleibt identisch, da sich das Element (inhaltlich) ja nicht verändert hat.

Wird beim erneuten Aufruf der `render`-Funktion ein anderes Element übergeben oder eines mit anderen Properties, löscht React hingegen nicht alle

bereits existierenden DOM-Elemente, sondern entfernt bzw. modifiziert nur diejenigen, die sich auch tatsächlich verändert haben.

Rufst du in dem oben gezeigten Beispiel die `render`-Funktion ein zweites Mal auf, übergibst aber ein anderes Label für die `Title`-Komponente, so wird React das umschließende `div`-Element der Header-Komponente unverändert lassen und nur den Text innerhalb des `h1`-Elements aktualisieren.

### 4.8.1 Fehler beim Rendern abfangen: Error Boundaries

Wenn beim Rendern deiner Komponente ein Fehler auftritt, zum Beispiel weil du auf ein Property zugreifst, das nicht existiert, wird React deine Anwendung komplett aus dem DOM entfernen und der Browser zeigt nur noch eine weiße Seite an. Die Motivation dafür, die ganze Anwendung und nicht nur die betroffene Komponente zu entfernen, besteht darin, dass React nicht sicherstellen kann, ob die von dem Fehler betroffenen Stellen in der Anwendung »nur« falsch dargestellt werden oder ob es dadurch zu weiteren (schwerwiegenden) Fehlern kommt.

Um Fehler beim Rendern abzufangen, kannst du *Error Boundaries* definieren. Das sind Komponenten, die eine spezielle API implementieren müssen, und die du als eine Art Try-catch-Handler um deine Komponenten legen kannst. Fehler, die dann beim Rendern in der Komponentenhierarchie unterhalb der Error-Boundary-Komponente auftreten, werden an den Error Boundary übergeben und können dort behandelt werden. Du kannst darin zum Beispiel eine Meldung an ein Monitoring-Backend schicken und dem Benutzer eine sprechende Fehlermeldung anzeigen.

Da Error Boundaries zwingend mit Klassenkomponenten entwickelt werden müssen, zeigen wir dir ein konkretes Beispiel dafür im Anhang B zu Klassenkomponenten.

Wenn deine Anwendung mit Create React App erzeugt wurde, wirst du während der Entwicklungszeit eine Fehlermeldung auch im Browser sehen, die auch den Stacktrace enthält. Diese Ansicht ist allerdings nicht mehr in der Anwendung vorhanden, wenn du einen Produktionsbuild erzeugst.

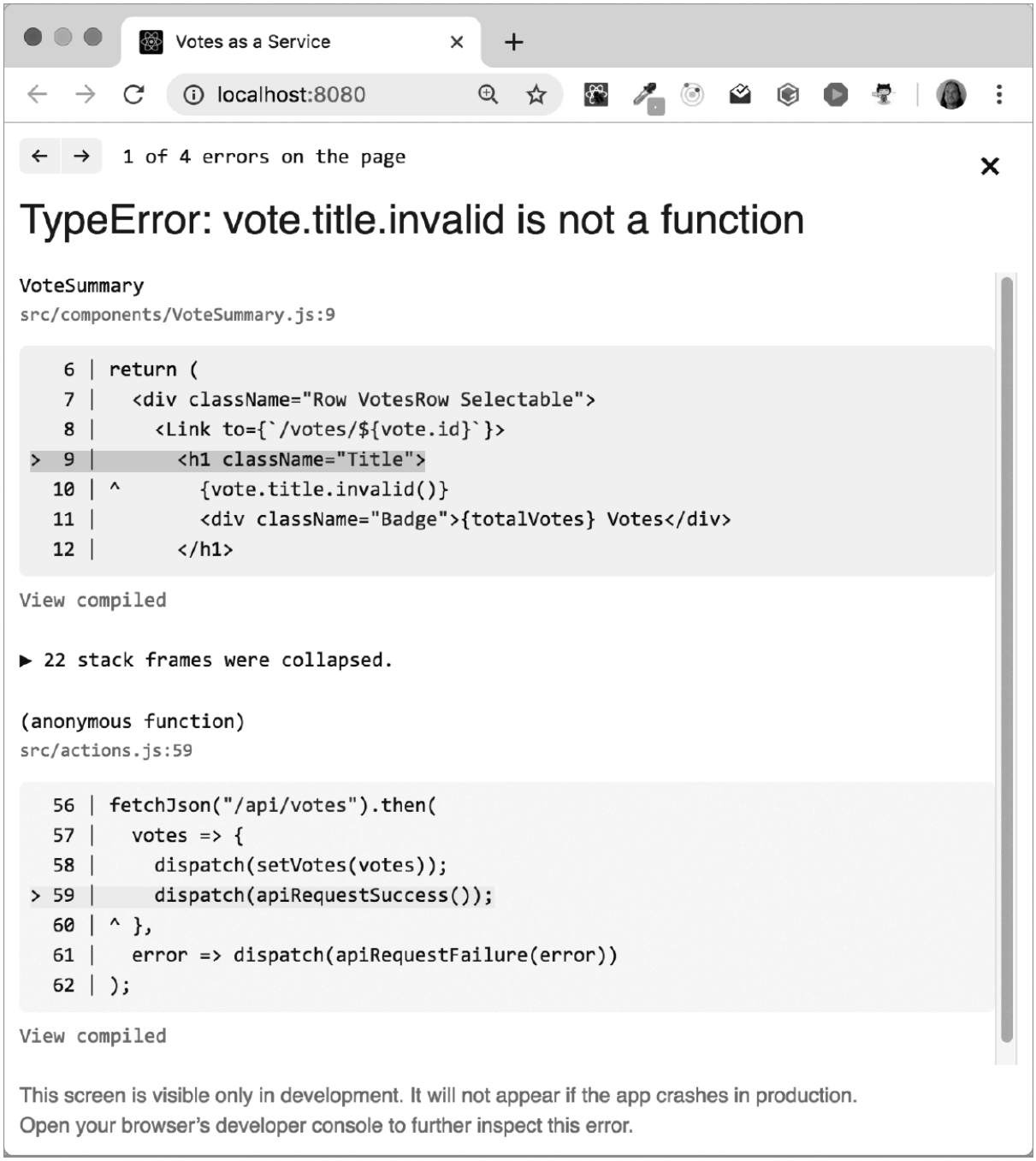


***Abb. 4–9*** *Ein Fehler beim Rendern wird im Browser dargestellt.*

**Tipp: Stacktrace mit Editor verknüpfen**

Du kannst auf eine Zeile in einem Stacktrace, der im Browser dargestellt wird, klicken und die entsprechende Codestelle wird unmittelbar in deinem Editor geöffnet. Create React App versucht, den von dir verwendeten Editor bzw. die IDE automatisch zu ermitteln. Falls das nicht klappt, kannst du den Editor bzw. die IDE mit der Umgebungsvariablen `REACT_EDITOR`[7], die du z.B. in der Datei `.env.local` im Projektverzeichnis setzen kannst, explizit angeben.

## 4.9 Arbeiten mit CSS

Das Styling deiner Komponenten kann über regulären CSS-Code (oder mit Präprozessoren wie Sass oder Less) erfolgen. Dabei gibt es mehrere Ansätze, wie du den Code in deine Anwendung einbindest. Unabhängig davon, für welchen Weg du dich entscheidest: Der CSS-Code, den du schreibst, ist unabhängig von React und hat keinerlei Bezug zu React. Du kannst ihn wie gewünscht strukturieren.

**Fertige Komponentenbibliotheken**

Wenn du kein eigenes Styling für deine Komponenten entwickeln möchtest, kannst du auch auf eine Vielzahl fertiger Komponentenbibliotheken für React zugreifen. So gibt es gleich mehrere Bibliotheken, die das Google Material Design implementieren, am weitesten verbreitet dürfte Material-UI sein[8]. Eine weitere sehr populäre, umfangreiche und anpassbare Bibliothek ist Ant Design[9]. Wenn du das CSS Bootstrap Framework in deinem Projekt verwenden möchtest, kannst du zum Beispiel React Bootstrap[10] verwenden.

Eine Liste von weiteren Komponentenbibliotheken findest du unter diesen URLs[11,12].

### 4.9.1 Plain CSS (manuelles Einbinden in index.html-Datei)

Du kannst bestehende oder neue CSS-Dateien wie aus HTML gewohnt in deiner `index.html`-Datei verlinken. Sofern du deine Anwendung mit Create React App erzeugt hast, kannst du das bestehende Template einfach erweitern.

### 4.9.2 CSS per import einbinden

Mit Webpack ist es grundsätzlich möglich, dass du deine CSS-(oder Less-/Sass-)Dateien auch aus JavaScript-Code heraus mittels der ES6-Import-Anweisung importierst. Je nach Art und Größe des CSS-Codes wird Webpack diesen im Build in eine eigene – minifizierte – CSS-Datei schreiben oder direkt in die `index.html`-Datei einbinden. Natürlich kannst du deinen CSS-Code auch auf mehrere Dateien verteilen.

Wenn du Create React App verwendest, um deine Anwendung zu erzeugen, ist dieser Ansatz bereits für dich konfiguriert und es wird auch empfohlen, ihn zu verwenden. Zum einen werden während des Builds auf diese Weise noch Optimierungen für den CSS-Code vorgenommen (insbesondere Minifizierung). Zum anderen musst du deinen Sourcecode nicht verlassen, um das Importieren von CSS-Dateien zu steuern. Wenn du Sass verwenden möchtest, musst du zuvor noch das Modul *node-sass* deinem Projekt hinzufügen. Dann kannst du ein oder mehrere CSS-Dateien in deiner Anwendung importieren.

```
import React from “react”;

import “./styles.css”; // CSS einbinden

import “./styles.scss”; // Sass-Datei einbinden
```

Die CSS-Klassen, die du in deinen Dateien verwendest, bleiben auf diese Weise wie von CSS gewohnt, global. Du hast also per se keine komponentenspezifischen CSS-Klassen. Das bieten dir aber die beiden folgenden Ansätze, CSS Modules und CSS-in-JS.

### 4.9.3 CSS Modules

Eine Variante des oben gezeigten Verfahrens sind CSS Modules[13]. Dabei teilst du deinen CSS-Code auf mehrere Dateien auf. Zur Build-Zeit werden die in den CSS-Dateien verwendeten Klassennamen durch eindeutige, generierte Namen ersetzt, sodass sich unterschiedliche Klassen, die in CSS global sichtbar sind, nicht gegenseitig beeinflussen. Du kannst auf diese Weise »isolierten« CSS-Code für jede einzelne Komponente schreiben.

CSS Modules sind React-unabhängig, Create React App ist aber für das Arbeiten mit CSS Modules konfiguriert. Genau wie oben gezeigt, importierst du auch in diesem Fall deine CSS- (oder Sass-)Dateien mit einem ES6-Import. Der Unterschied ist, dass du einen Namen für eine Variable beim Importieren angibst. Über diese Variable kannst du dann die Klassennamen referenzieren, und zwar über den Namen, so wie du ihn in deiner CSS-Datei angegeben hast. Zur Laufzeit wird er dann durch den eindeutigen, generierten Namen ersetzt. Um zu kennzeichnen, dass es sich bei einer CSS- oder Sass-Datei um eine CSS-Modules-Datei handelt, muss der Dateiname mit `.module.css` bzw. `.module.sass` oder `.module.scss` enden.

*Beispiel: CSS Modules*

Die `ChoiceBar`-Komponente könnte ihren spezifischen CSS-Code wie folgt per CSS Modules einbinden:

```
import React from “react”;

import styles from “./ChoiceBar.module.css”;

export default function ChoiceBar({ title, count, percent }) {
```

```
  return (
    <div className={styles.ChoiceBar}>
      <div className={styles.Progress}
           style={{ width: percent + “%” }}>
        <div className={styles.ChoiceBarTitle}>{title}</div>
      </div>
      <div className={styles.ChoiceBarBadge}>{count}</div>
    </div>
  );
}
```

Aus unserer Sicht ist die Verwendung von CSS Modules gerade in größeren Anwendungen eine sinnvolle Option, die auch weit verbreitet ist. Je nach Geschmack und Anforderungen kannst du diesen Ansatz problemlos mit Sass kombinieren. Möchtest du Less verwenden, musst du deine Create React App-Konfiguration leider zunächst manuell anpassen.

### 4.9.4 CSS-in-JS

Eine gänzlich andere Variante, die ebenfalls in der React-Entwicklung verbreitet ist, besteht darin, den CSS-Code direkt in den JavaScript-Code zu schreiben. Damit hast du alle Bestandteile deiner Komponente sogar inklusive Styling an einem Ort und kannst in deinen CSS-Styles auch auf dynamische Werte, etwa die Properties deiner Komponente, zugreifen. Obwohl der Ansatz mittlerweile Verbreitung findet, kann man ihn wohl (noch) nicht als Mainstream bezeichnen. Problematisch ist dieser Ansatz unter anderem in Fällen, wo das CSS zum Beispiel von Designern gebaut wird, die kein JavaScript-Code entwickeln können oder wollen.

### Styled Components

Der prominenteste Vertreter dieses Ansatzes für React ist die Bibliothek Styled Components[14].

*Beispiel: Komponenten mit Styled Components*

Mit dieser Bibliothek kannst du bestehende Komponenten, die »ungestylt« sind, mit festen CSS-Eigenschaften verknüpfen. Daraus erzeugt Styled Components neue »gestylte« Komponenten, die du innerhalb deiner Anwendung verwenden kannst. Aus einer `Button`-Komponente kannst du so zum Beispiel eine `RedButton`- und `BlackButton`-Komponente erzeugen.

```
import React from “react”;

import styled from “styled-components”;

const RedButton = styled.button`

  background-color: red;

`;

const BlackButton = styled.button`

  background-color: black;

  color: white;

`;
```

*Verwendung einer gestylten Komponente*

Die auf diese Weise erzeugten Komponenten sind reguläre React-Komponenten, die du wie gewohnt in deiner Anwendung verwenden kannst. Styled Components fügt den Komponenten zur Laufzeit automatisch CSS-Klassen hinzu, damit das von dir festgelegte Aussehen im Browser sichtbar wird.

```
import React from “react”;
```

```
function MyComponent() {

  return <div>

    <RedButton onClick={…}>OK</RedButton>

    <BlackButton onClick={…}>Cancel</BlackButton>

  </div>

}
```

*Stylings für eigene Komponenten*

Wenn du mit Styled Components aus »normalen« Komponenten »gestylte« Komponenten erzeugst, kannst du bei der Angabe der CSS-Eigenschaften auch auf die Properties der Komponente zugreifen und davon abhängig die Werte einer CSS-Eigenschaft setzen (siehe `Progress` unten[15]).

```
import React from “react”;

import styled from “styled-components”;

const ChoiceBarTitle = styled.div`

  left: 8px;

  top: 50%;

  font-size: 16px;

  …

`;

const ChoiceBarBadge = styled.div`  … `
```

```
const Progress = styled.div`

  width: ${props => props.value}%;

`;

function ChoiceBar({title, count, percent}) {

  return (

    <div>

      <Progress value={percent}>

        <ChoiceBarTitle>{title}</ChoiceBarTitle>

      </Progress>

      <ChoiceBarBadge}>{count}</ChoiceBarBadge>

    </div>

  );

}
```

In den oben gezeigten Beispielen haben wir jeweils nur die React-Komponenten für HTML-Elemente (`button`, `div`) mit Styles versehen. Auf eine ähnliche Weise kannst du auch CSS-Eigenschaften für deine eigenen Komponenten setzen. Dazu muss deine Komponente das Property `className` entgegennehmen und beim Rendern unverändert setzen. Dann erzeugst du mit der `styled`-Funktion aus deiner »ungestylten« Komponente eine »gestylte«, die du dann innerhalb deiner Anwendung verwendest:

```
function ChoiceBar({ title, count, percent, className }) {
```

```
  return (
    <div className={className}>
       … unverändert …
    </div>
  );
}
const StyledChoiceBar = styled(ChoiceBar)`
  border: 3px solid #ffb273;
  margin-bottom: 24px;
  …
  :hover { … }
  // Styling der Progress-Komponente innerhalb der ChoiceBar
  // anpassen
  :hover ${Progress} {
    background-color: #ff9640;
    cursor: pointer;
  }
```

```
  :hover ${ChoiceBarBadge} {…}

`;

export default StyledChoiceBar;
```

Die CSS-Klassen mit ihren Namen und Eigenschaften werden ausschließlich zur Laufzeit erzeugt und von Styled Components per JavaScript in ein `script`-Element am Anfang des DOM eingefügt. Du kannst das CSS also nicht bereits im Build erzeugen und in CSS-Dateien schreiben lassen.

(Dieses Beispiel findest du in `schritte/04b_eine_react_komponente_ styled_components` im Repository.)

## 4.10 Zusammenfassung

In diesem Kapitel haben wir das Herzstück von React gezeigt: Komponenten.

- Eine Komponente wird als Funktion oder ES6-Klasse implementiert. Da wir davon ausgehen, dass sich Funktionen künftig durchsetzen werden, stellen wir im Buch weitestgehend Funktionskomponenten vor.
- Der Name einer Komponente entspricht dem Namen der Komponentenfunktion. Sie gibt UI-Code zurück. Dazu wird die Spracherweiterung JSX verwendet, die es ermöglicht, HTML-artigen Code in JavaScript zu schreiben.
- Komponenten werden mit einem führenden Großbuchstaben benannt (`Title` statt `title`). Dadurch sind eigene Komponenten von den Standard-HTML-Elementen unterscheidbar.
- Verwender von Komponenten können einer Komponente Werte übergeben (Properties). Die Komponente darf diese Properties etwa zur Steuerung der Darstellung verwenden, aber nicht verändern.
- Für das Styling von Komponenten kannst du reguläres CSS verwenden. CSS kann auf mehrere Arten eingebunden werden. Auch populäre Ansätze wie CSS Modules und Präprozessoren wie Sass/Less werden unterstützt.

Im folgenden Kapitel zeigen wir dir, wie du Komponenten bauen kannst, die einen eigenen, veränderlichen Zustand haben und dadurch interaktiv sein können.

# Index

1-Wege-Databinding 125
2-Wege-Databinding 126

**A**

AbortController 153
absolute Importe 258
Action 132
Action (Redux) 219, 222
Action-Creators (Redux) 219
Alias (GraphQL) 381
all() (ES6 Promises) 357
Angular 15
Anwendungszustand 200
any (TypeScript) 375
Apollo (GraphQL) 287
Apollo CLI (GraphQL) 305
Apollo React Client (GraphQL) 287
Apollo-boost (GraphQL) 289
ApolloClient (GraphQL) 289
ApolloProvider (GraphQL) 289
Array (TypeScript) 372
Arrow Functions (ES6) 352
assign (ES6) 355
async/await 141, 358
Authentifizierung 198
Axios 137

**B**

Babel 9

Beispielanwendung (Votes as a Service)
Beispielcode 39
Choice-Objekt 37
Vote-Objekt 37
Benannte Exporte 351
BrowserHistory 195
Browserify 10
Build
für Produktion 31

## C

Cache 245
Redux 240
Cache (GraphQL) 297
Caching 96, 108
von Dateien 108
checkbox (HTML) 125
checked (HTML) 125
Children
einer Komponente 86
class (CSS) 65
className 47, 65
Code Coverage 175
codegen (Apollo GraphQL) 305
CodeSandbox 21
Code-Splitting 106
combineReducers 224, 225
mit TypeScript 238
Commit Phase 145
componentDidMount() 338
componentDidUpdate 338
componentWillUnmount 339
Concurrent Mode 109
const (ES6) 345
const assertion (TypeScript) 282
constructor (ES6) 349

Controlled Components 126
Datenfluss 126
Controller Components 91
Coverage Report 175
Create React App 13, 27, 195, 258
mit TypeScript 269
createRef 336
create-universal-react-app 320
CSS
Eigenschaften 48
importieren 49, 75
in JavaScript 77
in Komponenten 75
CSS Modules 76
CSS-Eigenschaften (JSX) 68
CSS-in-JS 77
Custom Elements 14
Custom Hooks 206, 239
Cypress 177

## D

Databinding 125
Deep -Links 191
defaultChecked 129
Defaultparameter (ES6) 354
DefinitelyTyped 376
DELETE (HTTP) 362
Destructuring (ES6) 347
Destructuring (TypeScript) 373

## E

ECMAScript 2015 10
End-to-End-Tests 177
Enzyme 179
Error Boundaries 73, 340
Erweiterte Objekt-Literale (ES6) 350
ES.next 9, 18

ES6 9
ESLint 29
Event Handler 335
    Methoden-Binding 335
expect (Jest) 166
export (ES6) 351
    default 351
externes Statemanagement 13
    mit Apollo GraphQL 303

## F

fetch 137, 141
    abbrechen 153
    GET (HTTP) 360
    json() 361
    mocken der Funktion 172
    POST (HTTP) 362
    text() 361
fetch-API 360
FetchPolicy (GraphQL) 298
fireEvent (React Testing Library) 169
First Page Impression 317
Flow 11
Flux 12
focus 131
focus() 130
for (CSS) 65
Formik 124
Formulare 111
Function as a Child 96

## G

Gatsby 320
Generatorfunktion (ES6) 359
GET (HTTP) 360
getByText (React Testing Library) 168
getState 245

Globaler Event Listener 62
gql-Funktion (GraphQL) 290, 305
GraphiQL 386
GraphQL 287
GraphQL Playground 288, 386
GraphQL Schema Language 384

## H

History-API 191
History-Objekt 197
Hooks 52, 54
    Regeln 56
    Übersicht 55
htmlFor 65
hydrate 315

## I

Immer 91
ImmutableJS 91
import (ES6) 351
input (HTML) 125
Integrationstests 177
interface (TypeScript) 370
Intersection Type (TypeScript) 373
Isomorphic Web Applications 319

## J

JavaScript
    Versionen 9
Jest 165
jest-dom 167
jest-fetch-mock 171
jsdom 177
JSX 63

## K

key (JSX) 67
Klassen (ES6) 348

Vererbung 349
Kommentare (JSX) 66
Kommunikation
zwischen Komponenten 93
Komponenten 4
Children 86
Controlled Components 126
CSS 75
deklarative 5
Hierarchie 81
Kommunikation 92
Konstruktor 338
Properties 50
refs 130
Rendern unterdrücken 98
Rückgabewerte 69
Uncontrolled components 128
Zustand 52
Komponentenbibliotheken 75
Konstruktor 330, 338

## L

Lazy Loading 106, 209
Lebenszyklus 59
einer Klassenkomponente 338
Less 75
let (ES6) 345
Link (Komponente) 191
Links (React Router) 191
Listen (JSX) 67
Lit Elements 15

## M

map() 84
Master-Detail-View 112
matchPath (React Router) 323
Material Design 75

memoization 97
Methoden-Binding 335
Middleware (Redux) 243
MobX 259
Module (ES6) 350
    exportieren 350
    importieren 351
mount (Enzyme) 181
Mutation (GraphQL) 380

## N

native DOM-Elemente 130
nativeEvent 62
Nebenläufigkeit 149
Next.JS 320
normalizr 91

## O

onBlur 129
onChange 127
onClick 193
option (HTML) 125

## P

pollInterval (GraphQL) 299
Portale 71
POST
    (HTTP) 362
Presentation Components 91
Prettier 12
Profiler 100
Progressive Enhancement 318
Progressive Web Apps 30
Promises (ES6) 356
Properties 50
    mit TypeScript 271
Puppeteer 177

PUT (HTTP) 362
PWA *Siehe* Progressive Web Apps

## Q

Query (GraphQL) 380

## R

radio (HTML) 125
razzle 320
React Context 203
    mit TypeScript 279
React Developer Tools 13, 100
React Native 16
React Router 12, 185
    BrowserHistory 195
    Hooks 193
    Integration mit Redux 252
    Links 191
    Redirects 197
    verschachtelte Routen 195
React Testing Library 167, 178
React.Component 329
React.createClass 330
React.createContext 204
React.createElement 64
React.createRef 336
    mit TypeScript 343
React.lazy 106, 209
React.memo 101, 240
React.PureComponent 334
React.RefObject 343
React.StrictMode 110
React.Suspense 106
react-app-polyfill 141
ReactDOM 45
    createPortal 71
    render 45, 59

ReactDOM.hydrate 315
ReactDOM.render 72
ReactDOMServer.renderToString 314
react-scripts 30
readQuery (GraphQL) 300
Redirects (React Router) 197
Reducer (Redux) 219, 223
Reducer-Funktion 132
Redux 13, 217
    Action mit TypeScript 282
    Integration mit React Router 252
    mit TypeScript 281
    Teilzustand 225
    useSelector 227
Redux Dev Tools 232
Redux Observable 248
Redux Saga 246
Redux Starter Kit 229
Redux Thunk 243
    mit TypeScript 284
ref-Callback-Funktion 131
refetch (GraphQL) 295
ref-Property 130, 131, 336
refs 130
reject() (ES6 Promises) 357
Relay (GraphQL) 287
Render Properties Pattern 94
render-Methode 329
Rendern 60
Render-Phase 145
renderToString 314
reselect 238
resolve() (ES6 Promises) 357
Rest Parameter (ES6) 354
ReturnType (TypeScript) 282

Route 193
Router *Siehe* React Router
RxJS 248

## S

Saga *Siehe* Redux Saga
Sass 75, 76
Schema (GraphQL) 384
Schema Definition Language (GraphQL) 384
Seiteneffekte 144, 150
    DOM 151
    in Klassenkomponenten 337
selected (HTML) 125
Selenium 177
SEO 317
Serverseitiges Rendern 311
setState 332
setState() 338
shallow (Enzyme) 179
Shallow Rendering (testen) 179
shallowEqual 236
shouldComponentUpdate 334, 338
simulate (Enzyme) 180
Single-Page-Anwendungen 3
Snapshot Testing 182
    React-Komponenten 183
Source Maps 269
SourceMaps 28
SPA *Siehe* Single-Page-Anwendungen
Spread-Operator (ES6)
    bei Objekten (ES2016) 355
    für Arrays 354
spyOn (Jest) 172
SSR *Siehe* Serverseitiges Rendern
State
    globaler 200

immutable 90
nach oben schieben 91
und Nebenläufigkeit 149
state 331
StaticRouter 314
Store (Redux) 219
Story Book 178
Strict Mode 109
Strukturelle Identität (TypeScript) 306
style (JSX) 68
Styled Components 77
Subscription (GraphQL) 380
Suspense 106, 209, 320
für Code-Splitting 106
zum Daten laden 162

## T

Template Strings (ES6) 346
Ternärer Operator (JSX) 66
TestCafe 177
Testen
React Add-on Test Utilities 181
textarea (HTML) 125
then (ES6 Promises) 356
this (Methoden-Binding) 335
Thunk
(Redux) 244
ThunkAction 285
Typdefinitionen generieren (GraphQL) 305
type (Action) 222
type (TypeScript) 370
Type Inference (TypeScript) 366
Type Narrowing (TypeScript) 369
__typename (GraphQL) 297
TypeScript 11, 263, 363
in Klassenkomponenten 341

## U

UI as a Function 5
Uncontrolled components 128
Union Type (TypeScript) 367
Universal Web Applications 311
urql (GraphQL) 287
useCallback 102, 234
useContext 206, 237
useDispatch 234
useEffect 144, 150, 234
useHistory 194, 197
useLocation 194
useMemo 96
useMutation (GraphQL) 295
    mit TypeScript 305
useParams 194
    mit TypeScript 280
useQuery (GraphQL) 291, 294
    mit TypeScript 304
useReducer 131, 219, 237
    mit TypeScript 275
useRef 129, 130
useSelector 226, 235
    mit TypeScript 283
useState 52, 58
    mit TypeScript 274
useSubscription (GraphQL) 302
    mit TypeScript 305

## V

value (HTML) 125
Virtueller Dom 64
Visual Regression Tests 178
Vue 16

## W

waitForElement (React Testing Library) 173
Web Components 14
Webpack 10, 108
Webpack Development Server 27
webpackChunkName 108
WHATWG 360
writeQuery (GraphQL) 300

## Y

yield (ES6) 359