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 zeigen. Am Ende dieses Kapitels entwickeln wir dann gemeinsam eine sehr einfache Komponente mit React, um dir auch einen ersten praktischen Eindruck des Frameworks zu vermitteln.

2.1 JSX

XML-Elemente im JavaScript-Code

JSX ist eine Spracherweiterung für JavaScript, die es dir erlaubt, XML-artige Elemente direkt im Code zu verwenden. Dadurch wird das Schreiben von Komponenten, in denen viel Code in Form von HTML-Elementen erforderlich ist, erheblich vereinfacht. Im Gegensatz zu anderen Frameworks benötigst du deshalb bei der Arbeit mit React keine explizite Template-Sprache. Stattdessen schreibst du deine Komponenten einfach mit JavaScript/JSX. Sieh dir dazu als Beispiel den folgenden Code an. Darin werden eine ganze Reihe von HTML-Elementen erzeugt:

Beispiel

```
function render() {
  return (
    <div>
      <h1>Hello!</h1>
      <p>What's your name?</p>
      <input type="text" autofocus name="username" />
    </div>
  );
}
```

React-API

Die gleiche Funktionalität könntest du auch mit der React-API in regulärem JavaScript schreiben, allerdings zeigt schon dieses kurze Beispiel, dass das viel aufwendiger und schwerer lesbar wäre:

```
React.createElement("div", null,
  React.createElement("h1", null, "Hello!"),
  React.createElement("p", null, "What's your name?"),
```

Oliver Zeigermann / Nils Hartmann, React, dpunkt.verlag, ISBN 978-3-86490-327-4

```
    React.createElement("input",
      {type: "text", autofocus: true, name: "username"}
    )
  )
```

Der Babel-Compiler

Die Konvertierung des JSX-Codes in diesen gültigen JavaScript-Code erfolgt durch einen Compiler. Bis zur Version 0.13 von React gab es dazu das React-eigene Tool JSTransformer. Dieses Tool ist mittlerweile eingestellt und so ist seit der Version 0.14 der Open-Source-Compiler Babel[1] das Tool der Wahl.Babel kann nicht nur JSX-Code übersetzen, sondern übersetzt auch ES6-Code nach ES5 und kann sogar bereits einige experimentelle Features der nächsten JavaScript-Version ES7 nach ES5 compilieren. Im Anhang des Buchs findest du ein Tutorial, in dem wir dir zeigen, wie du Babel in deinen Entwicklungsworkflow einbinden kannst.

Expressions

Es ist auch möglich, in JSX-Code normalen JavaScript-Code als Expressions zu verwenden, um beispielsweise dynamische Inhalte einzubinden. Expressions werden mit geschweiften Klammern umschlossen:

```
function getName() {
  return "Lemmy";
}

function render() {
  return (
    <div>
      <h1>Hello, {getName()}!</h1>
      <p>What's your name?</p>
      <input type="text" autofocus name="username" />
    </div>
  );
}
```

2.2 Komponenten

Zentrales Element von React sind Komponenten, die sowohl aus einem UI-Teil als auch aus Logik zur Steuerung der UI bestehen. Sie können über Eigenschaften (Properties) von außen konfiguriert werden und besitzen einen veränderlichen, internen Zustand (State). In dem veränderlichen Zustand werden beispielsweise Eingaben aus einem input-Feld abgelegt.

ES6-Klassen beschreiben wir im Anhang (Einführung in ES2015).

Technisch kann eine Komponente als Funktion oder als JavaScript-Klasse (ES6) geschrieben werden. In diesem Kapitel sehen wir uns zunächst nur die Klassen an, in Kapitel 5 beschäftigen wir uns

1. *http://babeljs.io/*

dann auch mit den Funktionen und zeigen dir die Unterschiede der beiden Varianten.

Komponenten als ES6-Klassen

Eine Komponentenklasse muss von `React.Component` abgeleitet werden und mindestens die Methode `render` implementieren, die die UI-Elemente zurückgibt, aus denen die Komponente besteht. Das folgende Beispiel zeigt eine sehr einfache, statische Komponente, die einen mit dem `<h1>`-Element umschlossenen String ausgeben würde. Die `render`-Methode verwendet wie oben beschrieben JSX:

Beispiel

```
import React from 'react';

class HeaderComponent extends React.Component {
  render() {
    return <h1>Hello, React!</h1>;
  }
}
```

Hinweis: React.createClass()

Die ES6-Klassennotation für Komponenten wird von React erst ab der Version 0.12 unterstützt. Zuvor wurden Komponentenklassen über die Funktion `React.createClass()` erzeugt. Dieses Konstrukt ist noch häufig in bestehenden React-Komponenten und -Artikeln zu finden. Für das Buch haben wir uns entschlossen, ausschließlich die neuere Variante zu verwenden, da wir auch möglichst viel ES6-Code verwenden möchten.

Exkurs: Virtueller DOM

Rendern der gesamten UI

Interessant dabei ist, dass immer die gesamte UI für einen Zustand erzeugt wird, die UI wird also nicht »inkrementell« beim Auftreten bestimmter Aktionen punktuell verändert. Aus der Entwicklung mit jQuery beispielsweise ist man es gewohnt, an einem DOM-Element einen Listener zu registrieren, z.B. um ein Input-Feld zu validieren. Je nach Zustand des Input-Felds werden dann direkt einzelne Elemente des DOM manipuliert, etwa indem an einem Element eine CSS-Klasse hinzugefügt oder entfernt wird, um das Aussehen zu beeinflussen – und durch rote Schrift eine fehlerhafte Eingabe zu signalisieren.

In einer React-Komponente hingegen wird nach der Änderung des Zustands (etwa Eingabe eines Zeichens durch den Anwender) immer die *gesamte* UI der Komponente durch Ausführen der `render`-Methode neu erzeugt.

Virtueller DOM

Damit das schnell geht, werden die Operationen allerdings nicht direkt auf dem nativen DOM durchgeführt. Vielmehr arbeitet React zunächst auf einem »virtuellen DOM«, dem Virtual DOM. Dabei handelt es sich um sehr leichtgewichtige JavaScript-Objekte. Wird die `ren-`

Oliver Zeigermann / Nils Hartmann, React, dpunkt.verlag, ISBN 978-3-86490-327-4

der-Methode deiner Komponente aufgerufen, arbeitet ihr JSX auf diesem virtuellen DOM, der Code sieht allerdings sehr stark nach »richtigem« DOM aus:

```
render() {
  return <h1>Mein Titel</h1>;
}
```

Vergleich der DOMs

Damit React nun nicht den kompletten DOM neu aufbauen muss, vergleicht es den neuen virtuellen DOM, der durch Aufruf der render-Methode entstanden ist, mit der vorherigen, also der zurzeit aktiven Variante des virtuellen DOM. Aus den Unterschieden der beiden virtuellen DOMs generiert React Operationen, die es dann auf dem echten DOM ausführt, um diesen in den von dir beschriebenen Zustand zu bringen. Es werden also auf dem echten DOM immer nur die minimal notwendigen Änderungen durchgeführt, um aus dem vorherigen Zustand den neuen zu bekommen. Dadurch ist React sehr schnell.

Dieses Verfahren hat für uns den Vorteil, dass wir uns nicht selber damit beschäftigen müssen, wie wir von einem Zustand unserer UI zum nächsten gelangen. Wir beschreiben einfach den Zustand, den wir jetzt haben möchten, und React kümmert sich darum, dass dieser Zustand erreicht wird. Das vereinfacht sowohl die Entwicklung als auch das Testen.

2.2.1 Eigenschaften (Properties) und Zustand (State)

Verhalten und Aussehen einer Komponente lassen sich über zwei Wege beeinflussen.

Properties

Zum einen gibt es die Eigenschaften (Properties) einer Komponente. Dabei handelt es sich um Werte, die der Komponente beim Erzeugen von *außen* durch den Verwender der Komponente übergeben werden. So könnte z.B. das Label einer Komponente von außen per Property gesetzt werden. Properties sind innerhalb der Komponente dann unveränderlich, können allerdings von außen jederzeit neu gesetzt werden.

Zustand einer Komponente: State

Daneben gibt es noch den *Zustand* der Komponente, der in React *State* genannt wird. Dabei handelt es sich um ein Objekt, das nur innerhalb der Komponente sichtbar ist, dafür aber – im Gegensatz zu den Eigenschaften – von der Komponente selber verändert werden kann. Hierin könnte zum Beispiel der Inhalt eines Textfelds abgelegt werden oder nach der Validierung eines eingegebenen Textes eine Fehlermeldung, die angezeigt werden soll.

Änderungen von außen – an den Properties – oder aus der Komponente heraus – am State – bewirken in der Regel ein neues Rendern der Komponente.

2.2.2 Events

Auf Ereignisse reagieren

Bei der Entwicklung von UI-Komponenten ist es immer wieder erforderlich, auf bestimmte Ereignisse zu reagieren, zum Beispiel, wenn auf einem Element mit der Maus geklickt oder in ein Eingabefeld Text eingegeben wurde. Dazu sind für HTML-Elemente eine ganze Reihe von Events standardisiert, auf die auch innerhalb einer React-Komponente reagiert werden kann. In diesem Zusammenhang werden Listener an den Elementen registriert, die in der `render`-Methode der Komponente erzeugt werden. React sorgt intern dafür, dass die Events übergreifend in allen Browsern identisch funktionieren. In dem folgenden Beispiel würde beim Klicken auf die Überschrift (`H1`-Element) »Hello World« auf der Konsole des Browsers ausgegeben:

Beispiel: Event Handler

```
render() {
  return <h1 onClick={()=>console.log("Hello World")}>Hello!</h1>;
}
```

2.2.3 Styles und className

Das Stylen der Komponenten kann in React wie bei gewöhnlichen HTML-Elementen auf zwei Wegen passieren: Zum einen können auf einem Element CSS-Klassen gesetzt werden. Zum anderen lassen sich einem Element auch direkt konkrete CSS-Eigenschaften zuweisen. Sowohl Klassen als auch Styles können abhängig vom Zustand der Komponente gesetzt werden.

className

CSS-Klassen können an einem Element mit dem Attribut `className` gesetzt werden. Da das an HTML-Elementen normalerweise verwendete `class` ein Schlüsselwort in JavaScript ist, kann es an dieser Stelle nicht verwendet werden.

CSS-Eigenschaften eines Elements werden mittels des aus CSS bekannten Attributs »style« gesetzt. Als Wert für dieses Attribut erwartet React allerdings keinen String mit den Eigenschaften, sondern ein JavaScript-Objekt. Dieses Objekt enthält die zu setzenden CSS-Eigenschaften, wobei der Name einer CSS-Eigenschaft als Schlüssel und der entsprechende Wert als Wert in dem Objekt anzugeben sind. Die Namen der CSS-Eigenschaften werden dabei allerdings nicht mit einem »-« gekoppelt (»font-size«), da auch dies kein gültiger Bezeich-

Oliver Zeigermann / Nils Hartmann, React, dpunkt.verlag, ISBN 978-3-86490-327-4

ner in JavaScript wäre, sondern in CamelCase-Schreibweise notiert (`fontSize`).

2.3 Ein Grüßaugust: Unsere erste React-Komponente

Nachdem du nun die wichtigsten Eigenschaften von React gesehen hast, möchten wir dir in diesem Abschnitt mit der Entwicklung einer sehr einfachen Komponente auch einen ersten praktischen Eindruck von der Arbeit mit React vermitteln.

Die Beispielkomponente besteht aus einem Eingabefeld zur Eingabe einer Begrüßung und einem Button zum Absenden der Begrüßung. Da die fiktiven Begrüßungen einer Längenbeschränkung unterliegen, wird in einem Textfeld außerdem ausgegeben, wie viele Zeichen noch zur Verfügung stehen bzw. ob das Limit bereits überschritten ist. Die Begrüßung kann nur abgeschickt werden, wenn die maximale Länge nicht überschritten ist.

Schritt 1: Die HTML-Seite anlegen und React einbinden

Im ersten Schritt legen wir eine nahezu leere HTML-Seite an, auf der wir dann unsere Komponente entwickeln werden. Zunächst enthält die Seite nur die grundlegenden Elemente einer HTML-Seite, sie bindet aber über den `script`-Tag im Header schon das React-Framework ein. Außerdem binden wir den Babel-Transformer ein, der den JSX- und ES6-Code unserer Komponente direkt im Browser nach ECMAScript5 übersetzt.

Transformation im Browser nur zum Experimentieren geeignet!

Das Transformieren von Code direkt im Browser ist nur für schnelles Ausprobieren geeignet, damit du zunächst keinen aufwendigen Build-Prozess aufsetzen musst. Für produktive Anwendungen eignet sich das Verfahren nicht. Wir zeigen dir im Anhang, wie man Babel im Build-Prozess verwendet.

Das Beispiel-Repository

Das beschriebene Beispiel ist in den einzelnen Schritten in unserem Git-Repository vorhanden: *https://github.com/reactbuch/greeter-example*. Dort findest du für jeden der folgenden Schritte ein eigenes Unterverzeichnis mit dem jeweils fertigen Stand.

In dem Repository findest du auch die von uns eingesetzten Versionen von React und Babel. Du brauchst zum Ausprobieren des Beispiels also keine weiteren Bibliotheken herunterzuladen.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Greeter</title>
    <script src="libs/react.js"></script>
    <script src="libs/react-dom.js"></script>
    <script src="libs/browser.min.js"></script>
  </head>
  <body>
  </body>
</html>
```

Listing 2–1
Der Ausgangspunkt: index.html

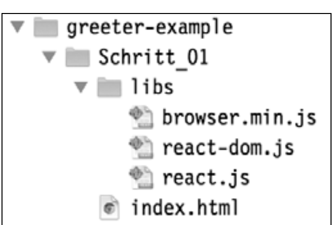


Abb. 2–1
Die Struktur des Beispielprojekts

Schritt 2: Initiale Version der »Greeter«-Komponente

Die Greeter-Komponente schreibst du direkt in die Datei `index.html`. Bitte beachte, dass das `type`-Attribut des umschließenden script-Elements auf den Wert `text/babel` (und nicht auf `text/javascript`) gesetzt werden muss, damit der darin enthaltene Code von Babel transformiert wird.

Die Komponente verfügt zunächst nur über eine `render`-Methode, in der die Elemente für das Eingabefeld, den Hinweistext und den Button zurückgegeben werden. Da die `render`-Methode einer React-Komponente allerdings immer nur genau ein Root-Element zurückliefern darf, wrappen wir diese drei Elemente in einem `div`-Element.

Nach der Definition der Komponentenklasse binden wir diese mit der React-Methode `ReactDOM.render` ein. Diese Methode erwartet als ersten Parameter die Komponente und als zweiten Parameter einen DOM-Knoten aus dem »echten« DOM, unterhalb dessen die Komponente eingehängt werden soll:

```
<!DOCTYPE html>
<html>
  <head>. . .</head>
  <body>
    <div id="greeterMountPoint"></div>
  </body>
  <script type="text/babel">
```

Listing 2–2
Die Greeter-Komponente

Oliver Zeigermann / Nils Hartmann, React, dpunkt.verlag, ISBN 978-3-86490-327-4

```
    class Greeter extends React.Component {
      render() {
        return (
          <div>
            Greeting:
            <input></input>
            <button>Greet</button>
          </div>
        );
      }
    }

    ReactDOM.render(
      <Greeter />,
      document.getElementById('greeterMountPoint')
    );
  </script>
</html>
```

Wenn du die HTML-Seite nun in einem Browser öffnest, siehst du die Elemente, die unsere Komponente erzeugt hat:

Abb. 2–2 *Die Greeter-Komponente im Browser*

Schritt 3: Auf Events und Zustandsverwaltung reagieren

Im nächsten Schritt erweitern wir unsere Komponente um das Behandeln von Events und die Zustandsverwaltung. Zum einen wollen wir nach jeder Eingabe eines Zeichens im Eingabefeld den dort eingegebenen Text im Zustand unserer Komponente abspeichern, damit wir daraufhin Gültigkeitsprüfungen ausführen können. Zum anderen soll der dann abgespeicherte Text ausgegeben werden, sobald der Benutzer auf den »Greet«-Button klickt.

Das state-Objekt

Das `state`-Objekt kann im Konstruktor der Komponentenklasse initial gesetzt werden. Dazu wird dort `this.state` auf ein beliebiges Objekt gesetzt. Auf diese Weise belegen wir in unserem Beispiel den Wert für unser Eingabefeld mit einem Leerstring vor. Zum späteren Verändern des Zustands wird die Methode `setState()` verwendet. Dadurch wird sichergestellt, dass React von der Zustandsänderung erfährt und das erneute Rendern der Komponente durch das Ausführen

der render-Methode durchführen kann. Lesend kann auf das Zustandsobjekt jederzeit über das Property state der Komponentenklasse zugegriffen werden. Das tun wir in unserer Komponente an zwei Stellen: Zum einen befüllen wir das Eingabefeld mit dem im state-Objekt hinterlegten Wert (Attribut value der input-Komponente). Zum anderen lesen wir diesen Wert, sobald der Greet-Button gedrückt wurde, um die eingegebene Begrüßung ausgeben zu können.

Event Handler

Um auf Ereignisse aus dem UI zu reagieren, kannst du an einem HTML-Element Event Handler registrieren. In unserer Komponente reagieren wir auf Änderungen im Eingabefeld (onChange) sowie auf das Auslösen des Buttons (onClick). Die jeweiligen Handler-Methoden werden als Funktionen an der Komponentenklasse implementiert und über eine ES6 Arrow Function aufgerufen[2].

Unsere überarbeitete Komponente sieht jetzt wie folgt aus:

Listing 2–3
Die Greeter-Komponente reagiert auf Ereignisse.

```
<script type="text/babel">
  class Greeter extends React.Component {
    constructor() {
      super();
      this.state = {
          greeting: ''
      }
    }

    onGreetingChange(event) {
      this.setState({
        greeting: event.target.value
      });
    }

    onGreetClicked() {
      // ES6 Template-String.
      // Backticks (`) statt Hochkomma (')
      alert(`Hello, ${this.state.greeting}`);
    }

    render() {
      return (
        <div>
          Greeting:
          <input value={this.state.greeting}
                 onChange={(e)=>this.onGreetingChange(e)}>
          </input>

          <button
            onClick={()=>this.onGreetClicked()}>Greet
          </button>
```

2. ES6 Arrow Functions und ES6 Template Strings stellen wir dir im Anhang vor (Einführung in ES2015).

Oliver Zeigermann / Nils Hartmann, React, dpunkt.verlag, ISBN 978-3-86490-327-4

```
        </div>
      );
    }
  }

  ReactDOM.render(
    <Greeter />,
    document.getElementById('greeterMountPoint')
  );

</script>
```

Wenn du diesen Stand in deinem Browser öffnest, kannst du in dem Eingabefeld eine Nachricht eingeben. Nach dem Klick auf *Greet* wird diese Nachricht dann ausgegeben.

Schritt 4: Die Eingabe validieren

In der jetzigen Version unserer Komponente können die eingegebenen Begrüßungen beliebig lang sein. Das Auslösen des Greet-Buttons funktioniert unabhängig von der Länge und er kann auch dann angeklickt werden, wenn überhaupt kein Text in das Eingabefeld eingegeben worden ist. Dieses Verhalten wollen wir in diesem Schritt unseres Beispiels ändern. Ziel soll sein, dass nur Begrüßungen erlaubt sind, die mindestens ein Zeichen umfassen und eine vorgegebene Maximallänge nicht überschreiten. Dem Benutzer soll außerdem angezeigt werden, wie viele Zeichen ihm noch zur Verfügung stehen.

Properties

Zunächst muss die Komponente dazu wissen, wie lang die Begrüßung maximal sein darf. Da die Maximallänge vom Verwender der Komponente vorgegeben werden soll, wird sie vom Verwender als Property an die Komponente übergeben (`maxLength`).

Mit der Maximallänge kann die Komponente dann in der `render`-Methode bestimmen, ob die eingegebene Begrüßung »gültig« ist, d.h. innerhalb der erlaubten Länge liegt. Dazu legen wir in der Komponente einige Hilfsmethoden an, die den eingegebenen Wert überprüfen (`getCharsRemaining`, `hasRemainingChars`, `isGreetingValid`).

Je nachdem, ob eine gültige Begrüßung eingegeben ist, soll die Komponente reagieren und den Greet-Button durch das Setzen des disabled-Attributs aktivieren bzw. deaktivieren. Außerdem kann die Komponente jetzt die Anzahl der noch erlaubten Zeichen berechnen und anzeigen.

Die Komponente sieht nun wie folgt aus. Bitte beachte, dass sich auch der Aufruf der Komponente geändert hat, da wir dort jetzt die vorgegebene Maximallänge übergeben:

```
class Greeter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      greeting: ''
    }
  }

  onGreetingChange(event) { . . . }

  onGreetClicked() { . . . }

  getCharsRemaining() {
    return this.props.maxLength - this.state.greeting.length;
  }

  hasRemainingChars() {
    return this.getCharsRemaining() > 0;
  }

  isGreetingValid() {
    return this.state.greeting.length > 0 &&
      this.getCharsRemaining() >= 0;
  }

  render() {
    const greetingValid = this.isGreetingValid();

    return (
      <div>
        Greeting:
        <input value={this.state.greeting}
               onChange={(e)=>this.onGreetingChange(e)}>
        </input>
        <span>{this.getCharsRemaining()}</span>

        <button disabled={!greetingValid}
                onClick={()=>this.onGreetClicked()}>Greet
        </button>
      </div>
    );
  }
}
ReactDOM.render(
  <Greeter maxLength={10}/>,
  document.getElementById('greeterMountPoint')
);
```

Listing 2–4
Die Greeter-Komponente validiert die Eingabe.

Schritt 5: Styling

Unsere Komponente lässt jetzt nur noch gültige Begrüßungen zu. Ob der eingegebene Wert gültig ist oder nicht, wollen wir aber auch noch visuell hervorheben. Dazu stylen wir die Komponente mit CSS. So sol-

Oliver Zeigermann / Nils Hartmann, React, dpunkt.verlag, ISBN 978-3-86490-327-4

len das Textfeld und der Zeichenzähler jeweils grün (gültige Begrüßung) oder rot (ungültige Begrüßung) dargestellt werden.

Um zu zeigen, wie du sowohl CSS-Klassen als auch Inline Styles verwenden kannst, stylen wir das Eingabefeld über CSS-Klassen (`valid` bzw. `invalid`). CSS-Klassen werden in React nicht mit dem HTML-Attribut `class`, sondern mit `className` gesetzt.

Für den Zeichenzähler setzen wir direkt an dem Element die entsprechenden CSS-Eigenschaften. Dazu wird in React einem Element mit dem `styles`-Property ein Objekt übergeben, das die gewünschten Styles enthält.

Sowohl die CSS-Klasse als auch die CSS-Eigenschaften müssen dabei natürlich abhängig vom jeweiligen Zustand der Komponente gesetzt werden.

Unser Beispiel sieht nun wie folgt aus:

Listing 2–5
Styling der Greeter-Komponente

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Greeter</title>
    <script src="libs/react.js"></script>
    <script src="libs/react-dom.js"></script>
    <script src="libs/browser.min.js"></script>
    <style>
      .valid {
        background-color: #a5d6a7;
      }
      .invalid {
        background-color: #ffcdd2;
      }
    </style>

  </head>
  <body>
    <div id="greeterMountPoint"></div>
  </body>
  <script type="text/babel">
    class Greeter extends React.Component {
      constructor(props) {
        super(props);
        this.state = {
          greeting: ''
        }
      }

      onGreetingChange(event) { . . . }

      onGreetClicked() { . . . }

      getCharsRemaining() { . . . }
```

```
      hasRemainingChars() { . . . }

      isGreetingValid() { . . . }

      render() {
        const greetingValid = this.isGreetingValid();

        return (
          <div>
            Greeting:
            <input className={greetingValid?'valid':'invalid'}
                   value={this.state.greeting}
                   onChange={(e)=>this.onGreetingChange(e)}>
            </input>
            <span style=
              {{color:this.hasRemainingChars()?'green':'red'}}>
                {this.getCharsRemaining()}
            </span>

            <button disabled={!greetingValid}
                    onClick={()=>this.onGreetClicked()}>Greet
            </button>
          </div>
        );
      }
    }

    ReactDOM.render(. . .);
  </script>
</html>
```

Schritt 6: Feinschliff

Mittlerweile verhält sich unsere Komponente so wie gewünscht. Ein letzter Feinschliff soll allerdings nicht fehlen: Nach dem Absenden der Begrüßung mit dem Greeter-Button soll das Eingabefeld zurückgesetzt bzw. geleert werden. Außerdem soll der Fokus, der noch auf dem Button steht, wieder in das Eingabefeld gesetzt werden.

Das Leeren des Eingabefelds ist mit den bereits beschriebenen Mitteln einfach zu erreichen: Wir setzen über das Zustandsobjekt (`state`) den Wert dafür auf einen Leerstring. React führt daraufhin die `render`-Methode aus, in der der (jetzt leere) Wert für das Eingabefeld aus dem Zustandsobjekt gelesen wird.

Zugriff auf native DOM-Elemente

Das Fokussieren des Eingabefelds funktioniert mit der DOM-Methode `focus`, die auf HTML-Elementen definiert ist. Allerdings arbeitest du in React in der Regel nicht direkt mit nativen DOM-Elementen, sondern auf dem virtuellen DOM. Für Fälle, in denen du doch auf einem nativen DOM-Element arbeiten musst, kannst du dir allerdings eine Referenz auf das native DOM-Element geben lassen. Dazu übergibst du dem entsprechenden Element mit dem Attribut ref eine Call-

Oliver Zeigermann / Nils Hartmann, React, dpunkt.verlag, ISBN 978-3-86490-327-4

back-Funktion, die aufgerufen wird, sobald das Element in den DOM gehängt wurde. Der Callback-Funktion wird dann von React das native DOM-Element übergeben. Dieses Element legen wir als Instanz-Variable in unserer Komponentenklasse ab, so dass wir in der `onGreetClicked`-Methode darauf Zugriff haben. Dort können wir dann die native DOM-Methode `focus()` aufrufen.

Nach den hierfür erforderlichen Änderungen an den `onGreetClicked`- und `render`-Methoden sieht unser fertiges Beispiel nun wie folgt aus:

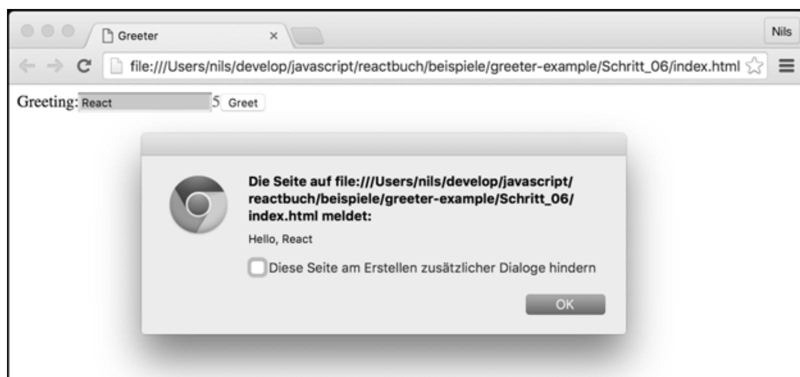
Listing 2–6
Die fertige Greeter-Komponente

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Greeter</title>
    <script src="libs/react.js"></script>
    <script src="libs/react-dom.js"></script>
    <script src="libs/browser.min.js"></script>
    <style>
      .valid {
        background-color: #a5d6a7;
      }
      .invalid {
        background-color: #ffcdd2;
      }
    </style>

  </head>
  <body>
    <div id="greeterMountPoint"></div>
  </body>
  <script type="text/babel">
    class Greeter extends React.Component {
      constructor(props) {
        super(props);
        this.state = {
          greeting: ''
        }
      }

      onGreetingChange(event) {
        this.setState({
          greeting: event.target.value
        });
      }

      onGreetClicked() {
        // ES6 Template-String.
        // Backticks (`) statt Hochkomma (')
        alert(`Hello, ${this.state.greeting}`);

        // Eingabefeld leeren durch zurücksetzen des States
```

```
      this.setState({
        greeting: ''
    });

      // Eingabefeld fokussieren
      // this.inputField ist natives DOM Element wenn gesetzt
    if (this.inputField) {
      this.inputField.focus();
    }
    }

    getCharsRemaining() {
      return this.props.maxLength - this.state.greeting.length;

    }
    hasRemainingChars() {
      return this.getCharsRemaining() > 0;
    }

    isGreetingValid() {
      return this.state.greeting.length > 0 &&
        this.getCharsRemaining() >= 0;
    }

    render() {
      const greetingValid = this.isGreetingValid();

      return (
        <div>
          Greeting:
          <input className={greetingValid?'valid':'invalid'}
                 value={this.state.greeting}
                 onChange={(e)=>this.onGreetingChange(e)}
                 ref={(c)=>this.inputField = c}
                 >
          </input>
          <span style=
            {{color:this.hasRemainingChars()?'green':'red'}}>
              {this.getCharsRemaining()}
          </span>

          <button disabled={!greetingValid}
                  onClick={()=>this.onGreetClicked()}>Greet
          </button>
        </div>
      );
    }
  }

  ReactDOM.render(
    <Greeter maxLength={10}/>,
    document.getElementById('greeterMountPoint')
  );
  </script>
</html>
```

Oliver Zeigermann / Nils Hartmann, React, dpunkt.verlag, ISBN 978-3-86490-327-4

Die fertige Greeter-Komponente gibt den eingegebenen Gruß aus.



2.4 Exkurs: Vergleich mit Angular

Wenn es um die Wahl des Frameworks für die eigene Single-Page-Anwendung geht, wird React häuft mit Angular verglichen. Aus diesem Grund zeigen wir dir im Folgenden die wichtigsten Gemeinsamkeiten und Unterschiede sowohl zu Angular 1 als auch zu Angular 2.

2.4.1 Angular 1

Angular kommt in zwei unterschiedlichen Versionen, die mehr oder weniger unterschiedlich und inkompatibel sind. Angular 1[3] gibt es seit einer ganzen Weile und man kann es ohne Wenn und Aber den »Hype vor React« nennen. Angular 1 ist ein komplettes Framework, inklusive Router und klarer Vorstellungen von einer Architektur und der Art und Weise, wie getestet wird. Der Kern von Angular 1 basiert auf den drei Ds: Doppeltes Data-Binding, Dependency Injection und Directives.

Doppeltes Data-Binding

Ein Modell kann an ein Eingabefeld gebunden werden, und wenn der Benutzer eine Eingabe macht, landet diese automatisch im Modell. Anders herum wird auch das Eingabefeld aktualisiert, wenn sich das Modell ändert. Dazu muss man außer der deklarativen Bindung nichts tun. Dieses doppelte Data-Binding ist sehr praktisch für Formulare.

React verzichtet ganz bewusst auf dieses doppelte Data-Binding und bindet nur in eine Richtung. Wie bereits erwähnt, führen Ände-

---

3. *https://angularjs.org/*

rungen am Modell (Zustand bei React) zur erneuten Darstellung eines Komponentenbaums. Für den Rückweg, also die Aktualisierung des Zustands, musst du Event Handler schreiben.

Technisch wäre es kein Problem, auch doppelt zu binden, und es gibt sogar Add-ons, die dies tun. Allerdings werden Anwendungen schwerer durchschaubar, wenn jede Komponente ihr eigenes Modell hat, das zudem noch automatisch verändert wird. Daher hat man bei React darauf verzichtet.

Depdendency Injection

Das Konzept von Depdendency Injection ist der Java-Welt entliehen. Man gibt über einen Namen an, welche Dienste man braucht, und Angular entscheidet, was für ein Objekt man bekommt. Die Abhängigkeit wird einem also von Angular injiziert. Das ist besonders sinnvoll bei Tests, wenn man statt eines echten Angular-Dienstes eine gemockte Version haben möchte, zum Beispiel um nicht wirklich auf einen Server zuzugreifen.

In React gibt es dieses Konzept nicht. Das Thema Tests wird bis auf ein Add-on ganz ausgeklammert. Das ist auch gut so, da andere Frameworks dies bereits sehr gut können. Diese Frameworks verzichten meist auf Dependency Injection, die in der JavaScript-Welt außerhalb von Angular so gut wie keine Verbreitung hat.

Directives

Das letzte D steht für Directives, also Angulars Weg, eigene HTML-Tags zu definieren. Dies wäre grundsätzlich vergleichbar mit Reacts Komponenten.

Da Angular direkt in das DOM des Browsers rendert und versucht, dies effizient zu tun, ist es nicht immer einfach, bestehende Direktiven zu verstehen oder selbst welche zu schreiben. In der Praxis führt dies dazu, dass ein komponentenbasierter Ansatz in Angular nicht verbreitet ist. Dies ist bei React ganz anders. Ohne Komponenten geht bei React nichts.

2.4.2 Angular 2

Angular 2[4] bricht mit einer ganzen Reihe von in Angular 1 getroffenen Designentscheidungen. Überraschenderweise ist Angular 2 dabei React deutlich näher als Angular 1. Teilweise arbeiten die Entwickler

4. *https://angular.io/*

Oliver Zeigermann / Nils Hartmann, React, dpunkt.verlag, ISBN 978-3-86490-327-4

von React und Angular 2 sogar zusammen, um z.B. Ideen für eine Rendering-Architektur auszutauschen. Dabei haben die Angular-2-Entwickler an der Performance gearbeitet und sich auch an bestehenden Best-Practices orientiert.

Data-Binding

Die auffälligste Änderung zu Angular 1 ist der Wegfall des doppelten Data-Bindings. Genauso wie bei React wird bei Angular 2 nur vom Modell zur Darstellung gebunden. Für den Rückweg musst du auch hier einen Handler angeben. Dies ist wie erwähnt näher an React als an Angular 1. Angular 2 enthält aber syntaktischen Zucker, der dich eine Form des 2-Wege-Bindings auch einfach ausdrücken lässt. So etwas unterstützt React über ein Add-on.

ES6 und TypeScript

Angular 2 ist mit TypeScript entwickelt worden und funktioniert am besten mit TypeScript- oder ES6/ES7-Syntax. Es geht aber auch ohne. React ist zwar intern immer noch in ES5 geschrieben, empfiehlt für die Programmentwicklung aber ebenso ES6/ES7-Syntax und liefert zum Teil sogar TypeScript-Deklarationen mit.

Komponenten

Bei Angular 2 sind Komponenten ebenso wie bei React zentral. Wenn man eine Angular-2-Anwendung möchte, muss man also eine oder mehrere Komponenten schreiben. Anwendungen werden ebenso wie bei React als Komponentenbäume umgesetzt. Allerdings wird die Darstellung über ein Template erreicht, das sogar in einer anderen Datei liegt und z.B. per ES7/TypeScript-Decorator der Komponente zugeordnet wird. Das ist deutlich angenehmer, wenn das Template nicht von derselben Person gepflegt wird, die den Rest der Komponenten geschrieben hat. Zudem gibt es über Komponenten eine Art eigene Template-Syntax für Abfragen und das Iterieren, also anders als in React, wo wir dafür nur JavaScript zur Verfügung haben.

Dependency Injection

Parameter über Konstruktor

Angular 2 bricht nicht mit der Idee der Dependency Injection. Allerdings wird nun nicht mehr über den Namen eines Parameters injiziert, was keine besonders gute Lösung war. Stattdessen werden die Abhängigkeiten deklariert und dann als Parameter in den Konstruktor einer Komponente hineingereicht.

Rendering

Trennung von interner und tatsächlicher Darstellung

Genau wie React trennt auch Angular 2 die interne Darstellungslogik von der tatsächlichen Darstellung, die typischerweise in der Aktualisierung des Browser-DOM besteht. Dadurch kann auch Angular 2 in unterschiedliche Ausgabemedien rendern und die interne Darstellungslogik ebenfalls in einem WebWorker ausführen. Dies führt zu einer Entlastung des Main-Threads und damit zu einer gefühlt schnelleren UI. Ebenso wie React ergibt sich hier aber auch die Möglichkeit, die Anwendung ganz anders darzustellen. Konkret sind native Plattformen wie Android und iOS im Gespräch. Dazu kommt serverseitiges Rendering, wie wir es bereits aus React kennen.

2.5 Zusammenfassung

In diesem Kapitel gaben wir dir einen Überblick über die Konzepte von React. Anhand des kurzen Beispiels konntest du eine erste Vorstellung davon bekommen, was es praktisch bedeutet, mit React Komponenten zu entwickeln:

- React-Komponenten können als ES6-Klassen implementiert werden.
- Komponenten können mit Properties konfiguriert werden und einen internen Zustand besitzen. Zur Interaktion können Komponenten Event Handler registrieren.
- Zum Beschreiben der UI-Elemente wird die JavaScript-Erweiterung JSX verwendet, die mit dem Babel-Compiler in reguläres JavaScript übersetzt wird.

In den folgenden Kapiteln stellen wir dir die entsprechenden Konzepte im Detail vor.

Oliver Zeigermann / Nils Hartmann, React, dpunkt.verlag, ISBN 978-3-86490-327-4