Table of contents

Vorwort

In diesem Buch geht es mir darum einen Einstieg zu ermöglichen, fortgeschrittene Themen aufzugreifen und dabei möglichst Best Practices zum Thema React zu vermitteln. Wenn ihr mit dem Buch nicht nur versteht wie etwas funktioniert, sondern auch warum, dann habe ich mein Ziel erfüllt. Nun hat jeder Entwickler andere Vorstellungen davon, welche Methoden die besten sind und wie man den einfachsten, effizientesten oder schönsten Code schreibt. Allerdings halte ich mich hier stark an die Empfehlungen von den Core-Entwicklern bei Facebook, die ebenfalls von der Community gut angenommenen Empfehlungen von AirBnB und noch einiger anderer Größen aus der „React-Szene". Alles gewürzt mit einer Prise eigener Erfahrung.

So gibt es bspw. mehrere Wege wie man seine Anwendung später publiziert, ob man es mit Tools wie **Browserify, Rollup** oder **Webpack** zu einem Bundle packt oder nicht. Ob man seine Komponenten als ES2015-Klassen schreibt oder `createClass` aus „ES5-Zeiten" verwendet. Dort wo ich es für sinnvoll erachte, werde ich auf die diversen gängigen Methoden eingehen um nicht nur Wege vorzugeben, sondern auch Alternativen aufzuzeigen.

Ich möchte hier jedoch primär möglichst auf die modernsten, aktuellsten und in den meisten Fällen auch einfachsten Methoden eingehen, weshalb ich für die meisten Code-Beispiele von einem Setup mit **Webpack**, **Babel** und **ES2015** (und neuer) ausgehen werde, das ich im weiteren Verlauf aber noch einmal sehr genau beschreiben werde. Wer zuvor nie mit ES2015+ in Berührung kam, wird sicherlich einen Augenblick länger benötigen, die Beispiele zu verstehen; ich werde mich indes bemühen, alle Beispiele verständlich zu halten und auch auf ES2015+ noch genauer eingehen. JavaScript-Grundkenntnisse sollten jedoch bei der Lektüre vorhanden sein.

Dieses Buch deckt außerdem nur das Thema **Einstieg in React** ab und bietet keinen Einstieg in JavaScript. Grundsätzliche und an einigen wenigen Stellen sicherlich auch etwas tiefergehende Kenntnisse in JavaScript werden daher vorausgesetzt, wobei ich alles möglichst einsteigerfreundlich erkläre, auch wenn man bisher nur einigermaßen oberflächlich mit JavaScript in Kontakt war. Ich setze nicht voraus, dass jeder Leser fehlerfrei erklären kann, wie ein JavaScript-Interpreter funktioniert, ich gehe aber sehr wohl davon aus, dass der Leser einigermaßen darüber Bescheid weiß, wie Scopes in JavaScript funktionieren, was ein Callback ist, wie `Promise.then()` und `Promise.catch()` funktionieren und wie das Prinzip asynchroner Programmierung mit JavaScript funktioniert.

Aber keine Sorge: das klingt komplizierter als es am Ende eigentlich ist. Jeder Leser, der in der Vergangenheit bereits bspw. mit jQuery gearbeitet hat, sollte beim größten Teil dieses Buches keine Verständnisprobleme haben und meinen Erklärungen folgen können.

Ein paar Worte zu diesem Buch

Ich habe dieses Buch im **Selbstverlag** veröffentlicht, da ich die volle Kontrolle über alle Veröffentlichungskanäle, das Preismodell, sowie sämtliche Rechte und Freiheiten über das Buch behalten möchte. Mir geht es nicht darum „den großen Reibach" zu machen, sondern ich möchte dieses Buch in erster Linie möglichst vielen Leuten zugänglich machen. Zu diesem Zweck gibt es eine kostenlose Online-Version dieses Buches, die ihr unter der URL https://lernen.react-js.dev/ findet. Mit dem Kauf der ePub oder PDF-Version über iBooks, Google Play Books, Amazon Kindle oder Leanpub kannst du meine Arbeit am Buch finanziell etwas unterstützen und bekommst statt der HTML-Version auch noch ein schönes E-Book zum Download.

Da **Selbstverlag** auch den Begriff „selbst" beinhaltet, bin ich aber auch komplett auf mich allein gestellt gewesen. Kein Verlag, der mir einen Lektor oder seine Vertriebsinfrastruktur zur Verfügung gestellt hat. Ein Lektor ist aber bereits beauftragt (stand Ende April), das dauert allerdings noch eine Weile bis dieser mit dem Lektorat fertig ist. Aus diesem Grund ist dieses Buch vielleicht an einigen Ecken nicht so rund wie man es von anderen Fachbüchern mit einem Verlag im Rücken gewohnt ist. Dafür bitte ich um Verzeihung. Solltet ihr einen Fehler finden, egal ob inhaltlich, grammatikalisch oder auch nur einen Tippfehler, könnt ihr mir jederzeit gern ein Ticket auf GitHub[1] erstellen.

Das Buch ist komplett im Markdown-Format geschrieben und liegt ebenfalls vollständig öffentlich auf GitHub[2]. Zum Schreiben habe ich Gitbook.com[3] verwendet. Grundsätzlich habe ich es geliebt, an einigen Tagen gehasst. Der Service eignet sich fantastisch zum Schreiben technischer Dokumentation, weniger tatsächlich um damit ganze Bücher zu schreiben, auch wenn der Name dies suggeriert.

Wenn euch das Buch gefällt oder ihr eine Frage habt, erreicht ihr mich am besten via Twitter. Hier freue ich mich über jede Rückmeldung, egal ob aufmunternde, positive Worte oder konstruktive Kritik!

Und nun wünsche ich euch viel Spaß mit dem Buch!

Über den Autor

Manuel Bieh, seit 2012 als Freelancer im Bereich Frontend-/JavaScript-Entwicklung tätig.

Bevor ich mich dazu entschied, als Freelancer zu arbeiten, hatte ich ebenfalls bereits fast 10 Jahre Erfahrung als Web-Entwickler in verschiedenen Unternehmen gesammelt, meist mit dem Fokus auf Frontend-Entwicklung. Lange Zeit habe ich mich eher als Generalist statt als Spezialist gesehen und so gab und gibt es wenige bekannte Frontend-Technologien, mit denen ich während meiner beruflichen Laufbahn nicht schon mal zumindest kurz in Berührung gekommen wäre. Als Spezialist habe ich mich aber (außer wenn es generell um JavaScript geht) nirgendwo gesehen. Dies änderte sich dann schlagartig, als mir ein befreundeter Entwickler in 2014 erstmals von React erzählte und ich dann durch Neugierde und ganz konkret durch ein Projekt für Zalando erstmals intensiver mit React in Kontakt kam.

Anfangs fremdelte ich noch etwas - so wie übrigens viele, die neu in React einsteigen, doch je länger und intensiver ich mich mit React auseinandersetzte, desto mehr schlug meine anfängliche Skepsis in Begeisterung um. Seitdem hat mich React so gepackt, dass ich meine Projekte allesamt so ausgewählt habe, dass dort React im Einsatz ist (und dessen Einsatz auch sinnvoll ist!). In dieser Zeit habe ich viel gelernt (und lerne auch immer noch jeden Tag dazu), habe dabei in kleinen Teams mit unter 5 und in recht großen Teams mit über 30 Leuten gearbeitet, dort mein React-Wissen eingebracht und selbst immer wieder neue Eindrücke und Wissen mitgenommen.

Die Komplexität von React ist dabei aber nicht zu unterschätzen. Und so ist es zwar möglich, relativ schnell in ziemlich kurzer Zeit eine Anwendung mit React zu entwickeln. Wenn man aber Wert auf hohe Qualität legt, gibt es dort viele Stellschrauben, an denen man drehen kann, um Code-Qualität, Performance und Wartbarkeit zu erhöhen; diese sind teilweise auch Leuten nicht bekannt, die schon viel und lange mit React entwickelt haben. Und so würde ich mich selbst nach mehrjähriger intensiver und täglicher Arbeit mit React sicher noch immer nicht als absoluten Experten bezeichnen. Aber ich denke, dass mit der Zeit dennoch genug Wissen zusammengekommen ist, welches ich in Form dieses Buches weitergeben kann, um euch den Einstieg zu erleichtern und auch noch den einen oder anderen Profi-Tipp an die Hand zu geben.

I – Einführung

Allgemeines

Was ist React eigentlich und was ist es nicht?

Zitieren wir hier an erster Stelle mal die React-Dokumentation, denn die bringt es sehr prägnant auf den Punkt:

> [React is] a library for building user interfaces.

Auch wenn die Erklärung sehr kurz ist, kann man aus ihr alle essentiellen Dinge ableiten, die für die Arbeit mit React wichtig sind und um zu verstehen worum es sich dreht. React ist erst einmal nur eine *Library*, kein vollständiges Framework mit unzähligen Funktionen, mit dem ihr ohne weitere Abhängigkeiten komplexe Web-Anwendungen entwickeln könnt. Und da kommen wir auch schon zum zweiten Teil des Satzes: *for building user interfaces*.

React ist also erst einmal lediglich eine **Library** die es euch einfach macht, **Benutzerinterfaces** zu entwickeln. Keine Services oder Methoden um API-Calls zu machen, keine built-in Models oder ORM. Nur User Interfaces. Sozusagen nur der View-Layer eurer Anwendung. That's it! In diesem Zusammenhang liest man gelegentlich, dass React das „V" in **MVC** (*Model-View-Controller*) oder **MVVM** (*Model-View-ViewModel*) darstellt. Das trifft es in meinen Augen ganz gut.

React bietet einen **deklarativen** Weg um den **State**, also den **Zustand** eines User Interfaces zu beschreiben. Vereinfacht gesagt bedeutet das, ihr beschreibt mit eurem Code im Grunde explizit wie euer User Interface aussehen soll, abhängig davon, in welchem **State** eine Komponente sich befindet. Einfaches Beispiel zur Veranschaulichung dieses Prinzips: ist ein Benutzer eingeloggt, zeige das Dashboard; ist er es nicht, zeige das Login-Formular.

Die Logik selbst befindet sich dabei komplett im JavaScript-Teil der Anwendung (dort, wo sie also immer hingehören sollte) und nicht in den Templates selbst, wie das bei den allermeisten anderen Web-Frameworks die Regel ist. Klingt erst einmal kompliziert, es wird aber im weiteren Verlauf immer deutlicher, was damit eigentlich gemeint ist.

React arbeitet dabei **komponentenbasiert**, d.h. man entwickelt **gekapselte, funktionale Komponenten**, die beliebig zusammengestellt (*composed*) und wiederverwendet werden können. Erweiterung bzw. Vererbung von Komponenten ist zwar theoretisch möglich, jedoch sehr unüblich in der React-Welt. Hier wird auch von offizieller Seite der *Composition-_Ansatz propagiert, bei dem mehrere Komponenten zu einem „Gesamtbild" zusammengefügt werden statt mit _Inheritance* (also Vererbung) zu arbeiten.

Bedeutet das jetzt also, dass ich keine komplexen Web-Anwendungen mit React entwickeln kann? Nein. Absolut nicht. React besitzt ein sehr großes, sehr aktives und zum großen Teil auch sehr hochqualitatives Ecosystem an Libraries, die wiederum auf React basieren, es erweitern oder ergänzen und so zu einem mächtigen Werkzeug werden lassen, das sich hinter großen Frameworks wie Ember oder Angular nicht verstecken muss. Im Gegenteil. Ist man erst einmal in die Welt des React-Ecosystems eingetaucht und hat sich einen Überblick verschafft, hat man ganz schnell eine Reihe an wirklich guten Tools und Libraries gefunden, mit denen man professionelle, super individuelle und hochkomplexe Anwendungen entwickeln kann.

Wann sollte ich React benutzen und wann nicht?

Insbesondere kurz nachdem React an Fahrt aufnahm, wurde oft die Frage gestellt, ob die Tage von jQuery nun gezählt sind - ob man nun alles mit React entwickeln kann oder gar soll oder wann der Einsatz von React sinnvoll oder vielleicht auch gar nicht sinnvoll ist.

React ist, wie wir bereits geklärt haben, erst einmal eine *Library für die Erstellung von User Interfaces*. User Interfaces bedeuten immer Interaktion. Und Interaktion geht zwangsweise in den meisten Fällen einher mit State-Management. Ich drücke einen Knopf und ein Dropdown öffnet sich. Ich ändere also den Zustand von *geschlossen* auf *offen*. Ich gebe Daten in ein Eingabefeld ein und bekomme angezeigt, ob meine eingegebenen Daten valide sind. Sind sie es nicht, ändert sich der Zustand des Eingabefeldes von *gültig* in *ungültig*. Und genau hier kommt React ins Spiel. Habe ich keine Interaktion oder „sich ändernde Daten“ auf meiner Seite, weil ich etwa eine reine statische Image-Seite für ein Unternehmen entwickle, brauche ich *wahrscheinlich* kein React.

Falsch umgesetzt kann React hier sogar schaden, da auf einer Image-Website oftmals der Content im Vordergrund steht und sofern man seine React-Komponenten nicht bereits serverseitig vorrendert, können die meisten Suchmaschinen mit der Seite erst einmal wenig anfangen. React macht es uns aber glücklicherweise sehr einfach, unsere Komponenten serverseitig zu rendern, von daher ist das noch ein Problem, welches sich in der Regel leicht beheben lässt.

Habe ich hingegen sehr viel Interaktion und ein Interface, das sich oft aktualisiert, wird der Einsatz von React mit ziemlich hoher Wahrscheinlichkeit sehr viel Zeit und Nerven sparen. Grundsätzlich gilt hier die Faustregel: je mehr Interaktion in einer Website oder Web-Anwendung stattfindet und je komplexer diese ist, desto mehr lohnt sich der Einsatz von React. Das griffigste Beispiel sind hier **Single Page Applications** (*SPA*), bei denen die Anwendung nur einmal im Browser aufgerufen und initialisiert wird und jegliche weitere Interaktion und Kommunikation mit dem Server über fetch-Requests oder XHR (den meisten besser bekannt als „AJAX-Requests“) abläuft.

Ich habe es kürzlich selbst in einem Projekt erlebt, dass ich ein Anmeldeformular entwickeln musste, welches mir ziemlich simpel erschien und ich startete erst einmal ohne React. Im Laufe der Entwicklung stellte sich heraus, dass zum Zwecke besserer Usability immer mehr (Hintergrund-)Interaktion nötig wurde. So sollte bspw. nachträglich eine automatische Live-Validierung von Formulardaten eingebaut und der Anmeldeprozess in 2 Schritte unterteilt werden, so dass ich recht zügig dann doch auf React zurückgegriffen habe, weil mir das manuelle State-Management und die **imperative** Veränderung des User Interfaces einfach zu umständlich wurde.

Imperativ bedeutet in dem Fall, dass ich dem Browser sage, was er machen soll, wohingegen ich bei *deklarativem* Code, wie man ihn mit React schreibt, lediglich das gewünschte Endergebnis anhängig vom aktuellen Zustand beschreibe. Eines der Kernprinzipien von React. Um beim Beispiel von oben zu bleiben: statt zu sagen „ich bin nun eingeloggt, lieber Browser, bitte blende nun das Login-Formular aus und zeige mir das Dashboard“, definiere ich zwei Ansichten: So, lieber Browser, soll mein Interface aussehen wenn ich eingeloggt bin (Dashboard-Ansicht) und so, wenn ich es nicht bin (Login-Ansicht). Welche der Ansichten angezeigt wird, entscheidet dann React anhand des Zustands der Komponente.

Wo hat React seinen Ursprung?

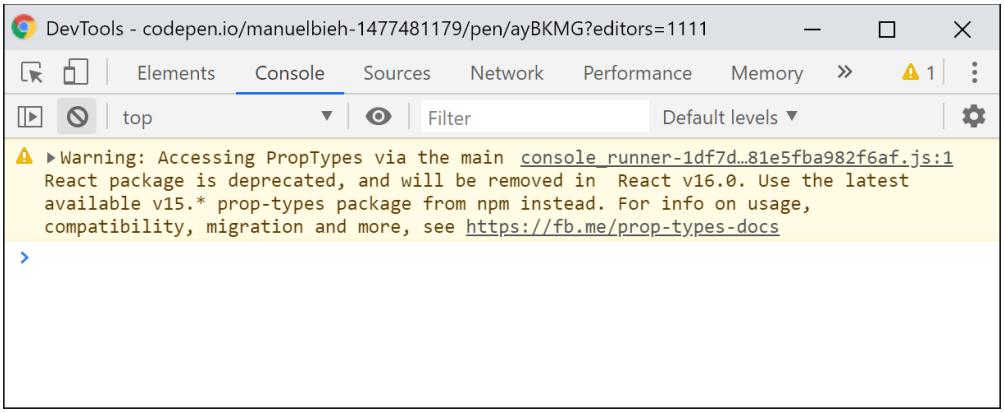
React wurde ursprünglich von bzw. bei **Facebook** entwickelt und später dann, bereits 2013, unter der BSD-Lizenz als Open Source der Öffentlichkeit zugänglich gemacht, die nach einigen Protesten in eine MIT-Lizenz geändert wurde. Und so basiert auch ein sehr großer Teil von Facebook auf React. Mittlerweile sollen sich dort sogar über **50.000** eigene Komponenten im Einsatz befinden - was insofern schön ist, als dass Facebook dadurch natürlich ein großes Interesse an der permanenten Weiterentwicklung hat und man nicht befürchten muss, dass man seine Anwendung auf Basis einer Technologie entwickelt hat, die plötzlich nicht mehr weiterentwickelt wird.

Die React Core-Entwickler leisten dabei sehr gute Arbeit darin, die Community frühzeitig in Entscheidungen miteinzubeziehen und mitdiskutieren zu lassen. Eigens dazu gibt es ein GitHub-Repository mit React RFCs[4] (*„Request for Comments“*), mittels dessen geplante Änderungen frühzeitig zur Diskussion gestellt werden und mittels dessen dem React-Team auch eigene Vorschläge unterbreitet werden können.

Breaking Changes, also Änderungen, die **nicht abwärtskompatibel** sind, folgen einem festen **Deprecation Schema** und so werden Methoden, Eigenschaften und Funktionen, deren Entfernung geplant ist, erst einmal für einige Zeit mit aussagekräftigen **Deprecation Warnings** versehen und es werden sogar Tools bereitgestellt, mit denen sich alter Code weitestgehend automatisiert anpassen lässt (React-Codemod[5]). React hält sich hier strikt an Semver-Konventionen.

Dies bedeutet, dass nur neue *Major-Releases* (`16.x.x` auf `17.x.x`) Breaking Changes enthalten, *Minor-Releases* (bspw. `16.6.x` auf `16.7.x`) enthalten neue Features oder bekommen Deprecation Warnings, die den Entwickler auf kommende Major-Releases vorbereiten, während *Patch-Releases* (bspw.`16.8.0` auf `16.8.1`) lediglich Bugfixes beinhalten.

Vor dem Release von Major- oder Minor-Releases gibt es regelmäßig auch Alpha-, Beta- und RC- (*„Release Candidate"*) Versionen, mit denen man vorab schon einen Blick auf kommende Features werfen kann. Diese sind aber jeweils mit Vorsicht zu genießen, da sich die Funktionsweise neuer Features bis zum endgültigen Release noch ändern könnten.



Beispiel für eine Deprecation Warning

Dies ist sicher dem Umstand geschuldet, dass eben auch bei Facebook sehr viele React-Komponenten im Einsatz sind und man dort nicht einfach mal eben tiefgreifende Änderungen vornehmen kann, ohne Probleme zu verursachen. Die Gedanken und Begründungen der Entwickler lassen sich dabei jederzeit ausführlich im GitHub Issue-Tracker verfolgen, alle wichtigen Änderungen werden dabei in sog. Umbrella-Tickets[6] zusammengefasst.

Ab ins kalte Wasser

Nun hatten wir bereits das „Was“, das „Wann“ und das „Wo“. Kommen wir also zum „Wie“ und schreiben unsere erste kleine **React-Komponente**. Neben **React** selbst benötigen wir für die Ausgabe unserer App im Browser auch das Package **ReactDOM,** um unsere Anwendung *mounten* zu können, also grob gesagt: im Browser nutzbar zu machen.

Ein sehr minimalistisches Setup, um schnell mit React loslegen zu können, sieht wie folgt aus:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>Hallo React!</title>
</head>
<body>
<div id="root"></div>
<script crossorigin src="https://unpkg.com/react@16.8.4/umd/react.development.js">
</script>
<script crossorigin src="https://unpkg.com/react-dom@16.8.4/umd/react-
dom.development.js"></script>
<script>
// Platzhalter für unsere erste Komponente
</script>
</body>
</html>
```

Wir erstellen also das Grundgerüst für ein gewöhnliches HTML-Dokument und laden **React** und **ReactDOM** in der jeweils aktuellsten Stable-Version vom unpkg-CDN, die uns dann jeweils als globale Variable im `window` Objekt unter `window.React` und `window.ReactDOM` zur Verfügung stehen. Ansonsten sehen wir hier vorerst nur eine leere Seite mit einem (noch inhaltlosen) `<div id="root"></div>`. Dieses div nutzen wir gleich als sogenannte **Mount-Node**, um dort unsere erste React-Komponente anzuzeigen.

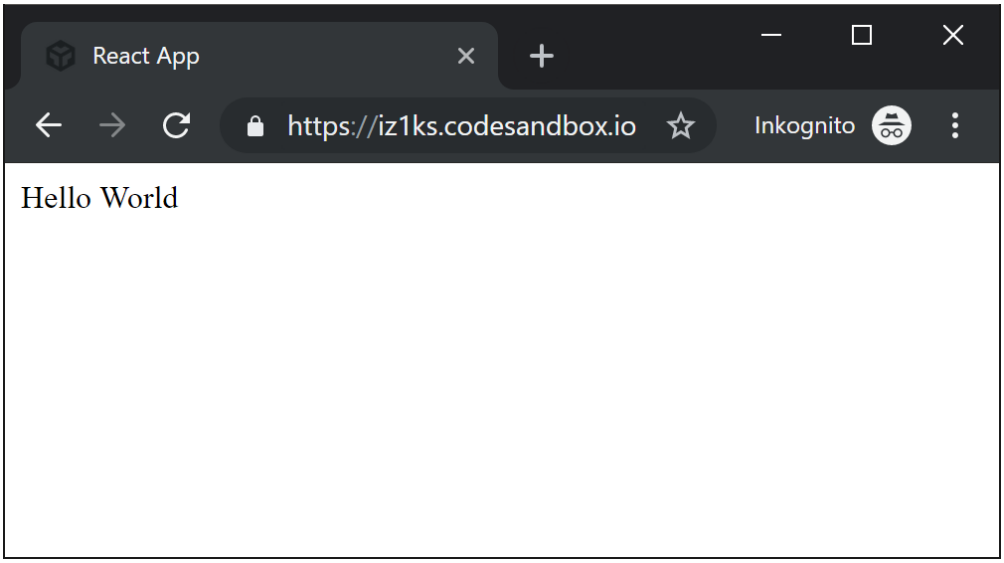
> ⓘ Sind mehrere React-Komponenten im Spiel, redet man üblicherweise von einer **App**, **WebApp** oder **Single Page App**. Die Grenzen, ab wann eine *Komponente* als *App* bezeichnet wird, sind dabei aber fließend. Einige Entwickler reden auch schon bei einer einzigen Komponente von einer **App**. Eine feste Definition gibt es dafür nicht.

Starten wir also klassischerweise mit dem üblichen „Hello World“-Beispiel und setzen das Script an die Stelle, an der sich oben der Platzhalter befindet:

```
class HelloWorld extends React.Component {
  render() {
    return React.createElement('div', { id: 'hello-world' }, 'Hello World');
  }
}

ReactDOM.render(
  React.createElement(HelloWorld),
  document.getElementById('root')
);
```

Und damit haben wir bereits die erste einfache React-Komponente implementiert! Setzen wir diesen Code nun an die Stelle unseres Platzhalters aus dem vorangegangenen Code-Snippet, sehen wir im Browser die folgende Ausgabe:



Unsere erste React-Komponente im Browser.

Sieht für’s Erste einmal gar nicht so kompliziert aus, oder? Gehen wir den Code einmal Schritt für Schritt durch. Die relevanten Stellen im Code habe ich fett hervorgehoben.

```
class HelloWorld
```

Hier geben wir dem Kind seinen Namen. Unsere Komponente hat in dem Fall den Namen **HelloWorld**. Bei der Namensgebung sind der Fantasie grundsätzlich keine Grenzen gesetzt, doch Achtung: React-Komponenten müssen stets mit einem Großbuchstaben beginnen! So

wäre helloWorld also kein gültiger Name für eine Komponente, HELLOWORLD hingegen schon (wenn auch sehr unüblich).

Die gängige Art der Benennung von Komponenten folgt der **UpperCamelCase**-Form. Auch längere, selbsterklärende Namen sind nicht unüblich. So wäre also ein Name wie **UserNotificationView** für eine Komponente keineswegs exotisch.

```
extends React.Component
```

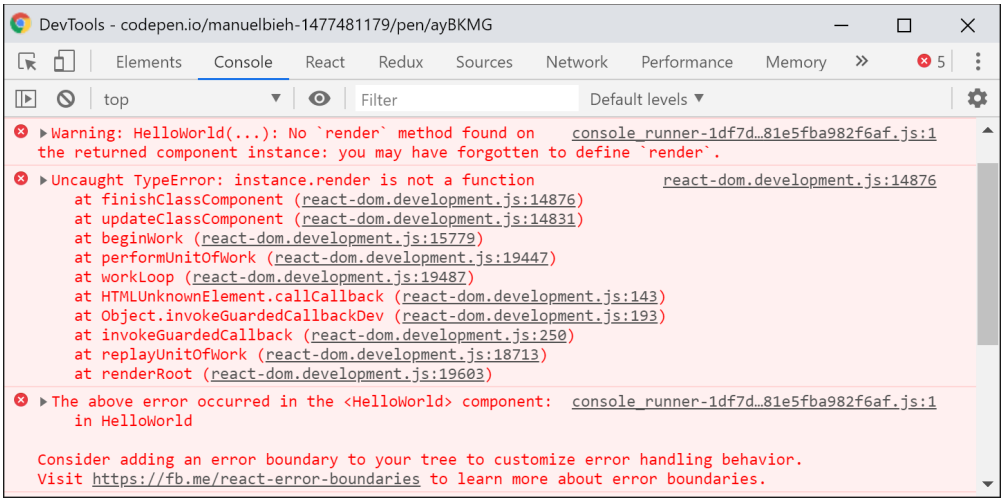
Hier erweitern wir schließlich die react-interne Klasse `React.Component`, wodurch unsere Klasse erst einmal zu einer Komponente wird, die wir in React nutzen können. Neben der `React.Component` gibt es außerdem auch die `React.PureComponent` als Komponenten-Klasse, sowie eine zweite Form, die sogenannte *Function Component*. Diese ist lediglich eine JavaScript-Funktion, die einem bestimmten Muster folgt. Beide werden im weiteren Verlauf noch ausführlich beleuchtet und sind an dieser Stelle zum Grundverständnis erst einmal weniger wichtig.

```
render();
```

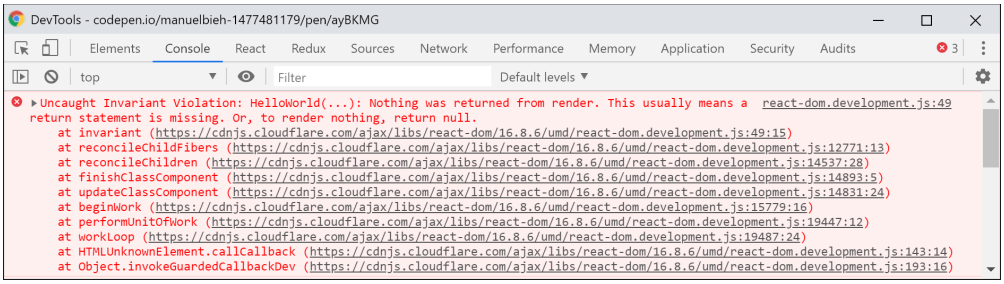
Unsere Komponente besteht lediglich aus dem einzigen zwingenden Bestandteil einer Komponente, nämlich der `render()`-Methode. Mittels dieser wird React mitgeteilt, wie die entsprechende Komponente dargestellt (sprich: *„gerendert“*) werden soll. Eine Komponente hat zwingend einen `return`-Wert. Dieser kann entweder ein explizites `null` sein, um bewusst nichts anzuzeigen (jedoch nicht `undefined`!), ein **React-Element** oder ab Version 16 auch ein **String** oder **Array**.

Im Falle eines Arrays darf dieser Strings, Numbers, React-Elemente oder ebenfalls `null` als Werte enthalten. Die `render()`-Methode dient also dazu, **deklarativ** den Zustand unseres Interfaces zu beschreiben. All das, was wir aus ihr per `return` zurückgeben, zeigt uns React beim Rendering als Ausgabe im Browser an.

Auch wenn man in der Gestaltung seiner JavaScript-Klassen natürlich vollkommen frei ist und dies daher nicht zwingend notwendig ist, so wird die `render()`-Methode der Übersicht halber in der Regel meist als letzte Methode einer Komponente definiert. So wird es etwa in den Code-Guidelines von AirBnB, dessen Entwickler in der React-Szene sehr aktiv sind, aber auch von vielen anderen bekannten Entwicklern vorgegeben oder zumindest empfohlen. Aus eigener Erfahrung kann ich sagen, dass es die tägliche Arbeit mit React deutlich erleichtert, sich an diese Empfehlung zu halten.



Fehlermeldungen bei fehlender render()-Methode



Fehlermeldung bei fehlerhafter render()-Methode

```
React.createElement();
```

Wie erwähnt gibt die `render()`-Methode einer React-**Komponente** in den meisten Fällen ein React-**Element** zurück. React-**Elemente** sind sozusagen die kleinsten aber dennoch gleichzeitig auch die wesentlichen Bausteine in einer React-Anwendung und beschreiben, was der Benutzer letztendlich auf seinem Bildschirm sieht. Neben `React.cloneElement()` und `React.isValidElement()` war `React.createElement()` sehr lange eine von lediglich 3 Top-Level API-Methoden.
Mittlerweile kamen einige weitere dazu, die aber vorrangig zur Performance-Optimierung dienen.

Die `createElement()`-Methode erwartet 1-n Parameter:

1. „Typ“, das können HTML-Elemente als String sein, also bspw. `'div'`, `'span'` oder `'p'` aber auch andere React-Komponenten
2. sog. „Props“, das sind im grundlegenden Sinn schreibgeschützte (*readonly*) „Eigenschafts-Objekte“ einer Komponente. Abgeleitet vom engl. *Properties* eben.

3. sowie beliebig viele Kind-Elemente, die selbst wieder React-Elemente, Arrays, Funktionen oder auch einfacher Text sein können. Eine Komponente muss aber nicht zwingend auch Kind-Elemente besitzen.

Letztendlich ist ein **React-Element** unter der Haube nichts weiter als ein unveränderliches (*immutable*) JavaScript-Objekt zur Beschreibung von Eigenschaften, die React mitteilen, wie etwas (und was) dargestellt werden soll. React erstellt nach dieser Beschreibung ein virtuelles Abbild der Komponenten-Hierarchie. Diese stellt eine Repräsentation des HTML-Baums in Form eines JavaScript-Objekts dar und wird manchmal auch noch als Virtual DOM bezeichnet, auch wenn das React-Team von dieser Bezeichnung eher wieder Abstand genommen hat. Dieser Baum wird anschließend von React dazu verwendet, möglichst nur die Teile einer Anwendung zu aktualisieren, in denen auch tatsächlich eine Änderung vorgenommen wurde, wenn der Benutzer mit der Anwendung interagiert, Daten verändert oder Events auslöst. Dazu wird grob gesagt der vorherige Baum mit dem aktuellen Baum verglichen.

Dadurch, dass React nicht einfach bei jeder State-Änderung die komplette Anwendung neu in den DOM schreibt, was aus Performance-Sicht sehr kostspielig wäre, sondern mittels eines **Reconciliation** (zu deutsch etwa *„Abgleich"*) genannten Prozesses zuvor vergleicht was geändert wurde, somit die Schreibvorgänge auf ein Minimum reduziert, wird ein zum Teil enormer Geschwindigkeitsvorteil erreicht gegenüber anderen Frameworks und Libraries die viele zum Teil unnötige DOM-Mutationen vornehmen.

Bei der täglichen Arbeit wird man `React.createElement()` jedoch für gewöhnlich niemals in dieser Form aufrufen, da uns **JSX**, eine von Facebook entwickelte Syntax-Erweiterung für JavaScript, diese Arbeit abnehmen und massiv erleichtern wird. Dennoch halte ich es für wichtig, von ihrer Existenz zu wissen um zu verstehen, wie **JSX** im Hintergrund arbeitet und so mögliche Fehlerquellen ausschließen zu können.

JSX sieht auf den ersten Blick aus wie HTML bzw. XML/XHTML, jedoch mit deutlich erweitertem Funktionsumfang und der Möglichkeit JavaScript-Ausdrücke darin zu verwenden. JSX ist eine Abstraktion um die Art, wie man React-Elemente erstellt, für den Entwickler **deutlich** zu vereinfachen. So würde unser obiges Beispiel:

```
React.createElement('div', { id: 'hello-world' }, 'Hello World');
```

in JSX ganz einfach wie folgt geschrieben werden:

```
<div id="hello-world">Hello World</div>
```

Was für viele Einsteiger in React erst einmal sehr befremdlich wirkt - ich habe in diesem Zusammenhang mal den schönen Begriff **JSX-Schock** gelesen - stellt sich aber nach etwas Rumspielerei jedoch sehr schnell als unglaublich praktisch heraus und ist meines Erachtens

einer der wesentlichen Gründe, warum React letztendlich so viel an Beliebtheit in so kurzer Zeit gewonnen hat.

Zurück zum Wesentlichen: unsere Komponente bekommt hier also über den `return`-Wert der `render()`-Methode mitgeteilt, dass sie ein Element vom Typ `div` mit der id `hello-world` und dem Kind-Element (in dem Fall ein Textknoten) mit dem Inhalt `Hallo Welt` darstellen soll.

```
ReactDOM.render(Element, Container);
```

Zu guter Letzt kommt mit `ReactDOM` die zweite Library ins Spiel. **ReactDOM** ist zuständig für das Zusammenspiel von React mit dem DOM (*Document Object Model*), also oberflächlich ausgedrückt: dem **Web-Browser**. Wie auch schon React selbst besitzt **ReactDOM** nur sehr wenige Top-Level API-Methoden. Wir konzentrieren uns vorerst mal auf die `render()`-Methode, die sozusagen das Herzstück von **ReactDOM** im Browser ist.

Trotz der Namensgleichheit hat diese erst einmal nicht direkt etwas mit der Methode innerhalb von React-Komponenten zu tun, sondern dient lediglich dazu, ein **React-Element** in eine angegebene **„Root-Node“** zu rendern, also stumpf ausgedrückt: anzuzeigen. In unserem Fall wird hier unsere `HelloWorld`-Komponente in das `<div id="root"></div>` gerendert. Die Root-Node wird dabei **nicht ersetzt**, sondern die Komponente wird **innerhalb des Containers** eingesetzt.

ReactDOM sorgt also dafür, **dass** wir die angegebene Komponente überhaupt erst einmal im Browser sehen können. **Was** wir dort genau sehen haben wir zuvor in der `render()`-Methode der Komponente über das angegebene React-Element als `return`-Wert beschrieben. Beim Aufruf von `ReactDOM.render()` wird dabei das als ersten Parameter angegebene **React-Element** in den als zweiten Parameter angegebenen **Container** gerendert.

Beim ersten Aufruf der `ReactDOM.render()` Funktion wird sämtlicher möglicherweise vorhandene Inhalt des Ziel-Containers durch den von React ermittelten, darzustellenden Inhalt ersetzt. Bei jedem weiteren Aufruf verwendet React einen internen Vergleichs-Algorithmus für bestmögliche Effizienz, um nicht die komplette Anwendung vollständig neu zu rendern!

In der Praxis ist das allerdings weniger von Relevanz, da die Funktion`ReactDOM.render()` bei der Erstellung von Single Page Apps üblicherweise nur einmalig ausgeführt wird, für gewöhnlich beim Laden einer Seite. React verändert dabei auch niemals den Ziel-Container selbst, sondern lediglich dessen Inhalt. Besitzt das Container-Element also eigene Attribute wie Klassen, IDs oder data-Attribute, bleiben diese auch nach dem Aufruf von `ReactDOM.render()` erhalten.

Damit ist das generelle Funktionsprinzip von React erst einmal erklärt, unsere erste Komponente ist implementiert und im Browser zu sehen!

Tools und Setup

Tools

Um störungsfrei und komfortabel mit React arbeiten zu können, sollten einige Bedingungen erfüllt sein. Nicht alles davon ist **zwingend** notwendig, es erleichtert das Entwicklerleben jedoch ungemein, weswegen ich dennoch **dringend** dazu rate und auch bei allen folgenden Beispielen davon ausgehen werde, dass ihr diese Tools installiert habt:

Node.js und npm

Node.js werden die meisten möglicherweise als **serverseitiges JavaScript** kennen, das ist allerdings nicht die ganze Wahrheit. In erster Linie ist **Node.js** einmal eine **JavaScript-Laufzeitumgebung**, die sich eben hervorragend für Netzwerkanwendungen eignet, also klassische Webserver. Darüber hinaus bringt **Node.js** auch ein Tool zur Paketverwaltung mit, nämlich **npm**, mit dem sich spielend einfach neue JavaScript-Libraries auf dem eigenen Rechner installieren lassen. Außerdem lassen sich auch eigene Kommandozeilen-Scripts damit schreiben und ausführen.

Statt **Node** direkt zu installieren, empfehle ich **nvm**[7] (*„Node Version Manager"*) für Mac und Linux bzw. **nvm-windows**[8] für Windows. **Nvm** hat den Vorteil, dass es einerseits keine Admin-Rechte benötigt, um Packages global zu installieren und man andererseits mit einem simplen Befehl auf der Kommandozeile (`nvm install [version]`) die auf dem System installierte Version aktualisieren kann. Für einer Liste aller verfügbaren Version kannst du ganz einfach `nvm ls-remote` (Mac/Linux) bzw. `nvm list available` (Windows) benutzen. Ich empfehle im weiteren Verlaufe dieses Buches, die aktuelle LTS (Long Term Support) Version zu benutzen. LTS Versionen sind stabile Versionen, die auch längere Zeit Updates erhalten.

Yarn

Während **Node** mit **npm** bereits einen guten und soliden Package-Manager mitbringt, geht **yarn** noch ein Stück weiter. Es bietet besseres Caching, dadurch auch bessere Performance, einfachere Kommandos und kommt darüber hinaus, wie React, ebenfalls aus dem Hause Facebook und wurde dort entwickelt, u.a. um die Arbeit mit React noch etwas angenehmer zu gestalten. Während alles, was hier im weiteren Verlauf des Buches beschrieben wird, auch mit **npm** ausgeführt werden kann, würde ich dennoch empfehlen, **Yarn** zu installieren. Dies gewinnt gerade im React-Umfeld mehr und mehr an Gewicht, insbesondere wegen seiner Einfachheit und seiner verbesserten Performance ggü. **npm**.

Exkurs ES2015+

Das „neue“ JavaScript

ES2015 ist kurz gesagt eine modernisierte, aktuelle Version von JavaScript mit vielen neuen Funktionen und Syntax-Erleichterungen. **ES2015** ist der Nachfolger von **ECMAScript** in der Version 5 (**ES5**), hieß daher ursprünglich auch einmal **ES6** und wird auch in einigen Blogs und Artikeln immer noch so bezeichnet. Stößt du also beim Lesen von Artikeln zu React auf den Begriff **ES6,** ist damit **ES2015** gemeint. Ich schreibe hier meist von **ES2015+** und meine damit Änderungen, die seit 2015 in JavaScript eingeflossen sind. Dazu gehören ES2016 (ES7), ES2017 (ES8) und ES2018 (ES9).

> ⓘ Das **ES** in **ES2015** und **ES6** steht für **ECMAScript**. Die ECMA International ist die Organisation, die hinter der Standardisierung der **ECMA-262-**Spezifikation steht, auf der JavaScript basiert. Seit 2015 werden jährlich neue Versionen der Spezifikation veröffentlicht, die aus historischen Gründen erst eine fortlaufende Versionsnummer beginnend ab Version 1 hatten, dann jedoch für mehr Klarheit die Jahreszahl ihrer Veröffentlichung angenommen haben. So wird **ES6** heute offiziell als **ES2015** bezeichnet, **ES7** als **ES2016**, usw.

Wer mit React arbeitet, nutzt in vermutlich 99% der Fälle auch **Babel** als **Transpiler,** um sein **JSX** entsprechend in `createElement()`-Aufrufe zu transpilieren. Doch **Babel** transpiliert nicht nur **JSX** in ausführbares JavaScript, sondern hieß ursprünglich mal **6to5** und hat genau das gemacht: mit **ES6**-Syntax geschriebenes JavaScript in **ES5** transpiliert, sodass neuere, zukünftige Features und Syntax-Erweiterungen auch in älteren Browsern ohne Unterstützung für „das neue“ JavaScript genutzt werden konnten.

Auf die meiner Meinung nach wichtigsten und nützlichsten neuen Funktionen und Möglichkeiten in **ES2015** und den folgenden Versionen möchte ich in diesem Kapitel eingehen. Dabei werde ich mich auf die neuen Funktionen beschränken, mit denen man bei der Arbeit mit React häufiger zu tun haben wird und die euch Entwicklern das Leben am meisten vereinfachen.

Wenn du bereits Erfahrung mit ES2015 und den nachfolgenden Versionen hast, kannst du dieses Kapitel überspringen!

Variablen-Deklarationen mit let und const

Gab es bisher nur `var` um in JavaScript eine Variable zu deklarieren, kommen in ES2015 zwei neue Schlüsselwörter dazu, mit denen Variablen deklariert werden können: `let` und `const`. Eine Variablendeklaration mit `var` wird dadurch in fast allen Fällen überflüssig, meist sind `let` oder `const` die sauberere Wahl. Doch wo ist der Unterschied?

Anders als `var` existieren mit `let` oder `const` deklarierte Variablen **nur innerhalb des Scopes ,in dem sie deklariert wurden!** Ein solcher Scope kann eine Funktion sein wie sie bisher auch schon bei `var` einen neuen Scope erstellt hat, aber auch Schleifen oder gar `if`-Statements!

Grobe Merkregel: überall dort, wo man eine öffnende geschweifte Klammer findet, wird auch ein neuer Scope geöffnet. Konsequenterweise schließt die schließende Klammer diesen Scope wieder. Dadurch sind Variablen deutlich eingeschränkter und gekapselter, was für gewöhnlich eine gute Sache ist.

Möchte man den Wert einer Variable nochmal überschreiben, beispielsweise in einer Schleife, ist die Variable dafür mit `let` zu deklarieren. Möchte man die Referenz der Variable unveränderbar halten, sollte `const` benutzt werden.

Doch Vorsicht: anders als bei anderen Sprachen bedeutet `const` nicht, dass der komplette Inhalt der Variable konstant bleibt. Bei Objekten oder Arrays kann deren Inhalt auch bei mit `const` deklarierten Variablen noch verändert werden. Es kann lediglich das Referenzobjekt, auf welche die Variable zeigt, nicht mehr verändert werden.

Der Unterschied zwischen `let`/`const` und `var`

Erst einmal zur Demonstration ein kurzes Beispiel dafür, wie sich die Variablendeklaration von `let` und `const` von denen mit `var` unterscheiden und was es bedeutet, dass erstere nur in dem Scope sichtbar sind, in dem sie definiert wurden:

```
for (var i = 0; i < 10; i++) {}
console.log(i);
```

Ausgabe:

> 10

Nun einmal dasselbe Beispiel mit `let`

```
for (let j = 0; j < 10; j++) {}
console.log(j);
```

Ausgabe:

> ⊗ Uncaught ReferenceError: `j` is not defined

Während auf die Variable `var i`, einmal definiert, auch außerhalb der `for`-Schleife zugegriffen werden kann, existiert die Variable `let j` nur innerhalb des Scopes, in dem sie definiert wurde. Und das ist in diesem Fall innerhalb der `for`-Schleife, die einen neuen Scope erzeugt.

Dies ist ein kleiner Baustein, der uns später dabei helfen wird unsere Komponenten gekapselt und ohne ungewünschte Seiteneffekte zu erstellen.

Unterschiede zwischen `let` und `const`

Folgender Code ist valide und funktioniert, solange die Variable mittels `let` (oder `var`) deklariert wurde:

```
let myNumber = 1234;
myNumber = 5678;
console.log(myNumber);
```

Ausgabe:

> ⓘ 5678

Der gleiche Code nochmal, nun allerdings mit `const`:

```
const myNumber = 1234;
myNumber = 5678;
console.log(myNumber);
```

Ausgabe:

> ⊗ Uncaught TypeError: Assignment to constant variable.

Wir versuchen hier also eine durch `const` deklarierte Variable direkt zu überschreiben und werden dabei vom JavaScript-Interpreter zurecht in die Schranken gewiesen. Doch was, wenn wir stattdessen nur eine Eigenschaft *innerhalb* eines mittels `const` deklarierten Objekts verändern wollen?

```
const myObject = {
  a: 1,
};
myObject.b = 2;
console.log(myObject);
```

Ausgabe:

> {a: 1, b: 2}

In diesem Fall gibt es keinerlei Probleme, da wir nicht die Referenz verändern, auf die die `myObject` Variable verweisen soll, sondern das Objekt, auf das verwiesen wird. Dies funktioniert ebenso mit Arrays, die verändert werden können, solange nicht der Wert der Variable selbst geändert wird!

Erlaubt:

```
const myArray = [];
myArray.push(1);
myArray.push(2);
console.log(myArray);
```

Ausgabe:

> [1, 2]

Nicht erlaubt, da wir die Variable direkt überschreiben würden:

```
const myArray = [];
myArray = Array.concat(1, 2);
```

> Uncaught TypeError: Assignment to constant variable.

Möchten wir `myArray` also überschreibbar halten, müssen wir stattdessen `let` verwenden oder uns damit begnügen, dass zwar der Inhalt des mittels `const` deklarierten Arrays veränderbar ist, nicht jedoch die Variable selbst.

Arrow Functions

Arrow Functions sind eine weitere **deutliche** Vereinfachung, die uns ES2015 gebracht hat. Bisher funktionierte eine Funktionsdeklaration so: man schrieb das Keyword `function`, optional gefolgt von einem Funktionsnamen, Klammern, in der die Funktionsargumente beschrieben wurden, sowie dem **Function Body**, also dem eigentlichen Inhalt der Funktion:

```
function(arg1, arg2) {}
```

Arrow Functions vereinfachen uns das ungemein, indem sie erst einmal das `function`-Keyword überflüssig machen:

```
(arg1, arg2) => {};
```

Haben wir zudem nur einen Parameter, sind sogar die Klammern bei den Argumenten optional. Aus unserer Funktion

```
function(arg) {}
```

würde also die folgende **Arrow Function** werden:

```
(arg) => {};
```

Jap, das ist eine gültige Funktion in ES2015!

Und es wird noch wilder. Soll unsere Funktion lediglich einen Ausdruck als `return`-Wert zurückgeben, sind auch noch die Klammern optional. Vergleichen wir einmal eine Funktion, die eine Zahl als einziges Argument entgegennimmt, diese verdoppelt und als `return`-Wert wieder aus der Funktion zurückgibt. Einmal in ES5:

```
function double(number) {
  return number * 2;
}
```

… und als ES2015 **Arrow Function**:

```
const double = (number) => number * 2;
```

In beiden Fällen liefert uns die eben deklarierte Funktion beim Aufruf von bspw. `double(5)` als Ergebnis `10` zurück!

Aber es gibt noch einen weiteren gewichtigen Vorteil, der bei der Arbeit mit React sehr nützlich sein wird: Arrow Functions haben keinen eigenen Constructor, können also nicht als Instanz in

der Form `new MyArrowFunction()` erstellt werden, und binden auch kein eigenes `this` sondern erben `this` aus ihrem **Parent Scope**. Insbesondere Letzteres wird noch sehr hilfreich werden.

Auch das klingt fürchterlich kompliziert, lässt sich aber anhand eines einfachen Beispiels ebenfalls recht schnell erklären. Nehmen wir an, wir definieren einen Button, der die aktuelle Zeit in ein `div` schreiben soll, sobald ich ihn anklicke. Eine typische Funktion in ES5 könnte wie folgt aussehen:

```
function TimeButton() {
  var button = document.getElementById('btn');
  var self = this;
  this.showTime = function() {
    document.getElementById('time').innerHTML = new Date();
  };
  button.addEventListener('click', function() {
    self.showTime();
  });
}
```

Da die als **Event Listener** angegebene Funktion keinen Zugriff auf ihren **Parent Scope**, also den **TimeButton** hat, speichern wir hier hilfsweise `this` in der Variable `self`. Kein unübliches Muster in ES5. Alternativ könnte man auch den Scope der Funktion explizit an `this` binden und dem **Event Listener** beibringen, in welchem Scope sein Code ausgeführt werden soll:

```
function TimeButton() {
  var button = document.getElementById('btn');
  this.showTime = function() {
    document.getElementById('time').innerHTML = new Date();
  };
  button.addEventListener(
    'click',
    function() {
      this.showTime();
    }.bind(this)
  );
}
```

Hier spart man sich zumindest die zusätzliche Variable `self`. Auch das ist möglich, aber nicht besonders elegant.

An dieser Stelle kommt nun die **Arrow Function** ins Spiel, die, wie eben erwähnt, `this` aus ihrem **Parent Scope** erhält, also in diesem Fall aus unserer `TimeButton`-Instanz:

```
function TimeButton() {
  var button = document.getElementById('btn');
  this.showTime = function() {
    document.getElementById('time').innerHTML = new Date();
  };
  button.addEventListener('click', () => {
    this.showTime();
  });
}
```

Und schon haben wir im **Event Listener** Zugriff auf `this` des überliegenden Scopes!

Keine `var self = this` Akrobatik mehr und auch kein `.bind(this)`. Wir können innerhalb des Event Listeners so arbeiten als befänden wir uns noch immer im `TimeButton` Scope! Das ist später insbesondere bei der Arbeit mit umfangreichen React-Komponenten mit vielen eigenen Class Properties und Methods hilfreich, da es Verwirrungen vorbeugt und nicht immer wieder einen neuen Scope erzeugt.

Neue Methoden bei Strings, Arrays und Objekten

Mit ES2015 erhielten auch eine ganze Reihe neue statische und Prototype-Methoden Einzug in JavaScript. Auch wenn die meisten davon nicht direkt relevant sind für die Arbeit mit React, erleichtern sie die Arbeit aber gelegentlich doch ungemein, weshalb ich hier ganz kurz auf die wichtigsten eingehen möchte.

String-Methoden

Hat man in der Vergangenheit auf `indexOf()` oder reguläre Ausdrücke gesetzt, um zu prüfen ob ein String einen bestimmten Wert enthält, mit einem bestimmten Wert anfängt oder aufhört, bekommt der String Datentyp nun seine eigenen Methoden dafür.

Dies sind:

```
string.includes(value);
string.startsWith(value);
string.endsWith(value);
```

Zurückgegeben wird jeweils ein Boolean, also `true` oder `false`. Möchte ich wissen ob mein String `Beispiel`ein `eis` enthält, prüfe ich ganz einfach auf

```
'Beispiel'.includes('eis');
```

Analog verhält es sich mit `startsWith`:

```
'Beispiel'.startsWith('Bei');
```

… wie auch mit `endsWith`:

```
'Beispiel'.endsWith('spiel');
```

Die Methode arbeitet dabei case-sensitive, unterscheidet also zwischen Groß- und Kleinschreibung.

Zwei weitere hilfreiche Methoden, die mit ES2015 Einzug in JavaScript erhalten haben, sind `String.prototype.padStart()` und `String.prototype.padEnd()`. Diese Methoden könnt ihr nutzen, um einen String auf eine gewisse Länge zu bringen, indem ihr am Anfang (`.padStart()`) oder am Ende (`.padEnd()`) Zeichen hinzufügt bis die angegebene Länge erreicht ist. Dabei gibt der erste Parameter die gewünschte Länge an, der optionale zweite Parameter das Zeichen mit dem ihr den String bis zu dieser Stelle auffüllen wollt. Gebt ihr keinen zweiten Parameter an, wird standardmäßig ein Leerzeichen benutzt.

Hilfreich ist das bspw. wenn ihr Zahlen auffüllen wollt, so dass diese immer einheitlich dreistellig sind:

```
'7'.padStart(3, '0'); // 007
'72'.padStart(3, '0'); // 072
'132'.padStart(3, '0'); // 132
```

`String.prototype.padEnd()` funktioniert nach dem gleichen Muster, mit dem Unterschied, dass es euren String am Ende auffüllt, nicht am Anfang.

Arrays

Bei den Array-Methoden gibt es sowohl neue statische als auch Methoden auf dem Array-Prototype. Was bedeutet dies? Prototype-Methoden arbeiten „mit dem Array“ als solches, also mit einer bestehenden **Array-Instanz**, statische Methoden sind im weiteren Sinne Helper-Methoden, die gewisse Dinge tun, die „mit Arrays zu tun haben“.

Statische Array-Methoden

Fangen wir mit den statischen Methoden an:

```
Array.of(3); // [3]
Array.of(1, 2, 3); // [1, 2 ,3]
Array.from('Example'); // ['E', 'x', 'a', 'm', 'p', 'l', 'e']
```

`Array.of()` erstellt eine neue Array-Instanz aus einer beliebigen Anzahl an Parametern, unabhängig von deren Typen. `Array.from()` erstellt ebenfalls eine Array-Instanz, allerdings aus einem „array-ähnlichen" iterierbaren Objekt. Das wohl griffigste Beispiel für ein solches Objekt ist eine `HTMLCollection` oder eine `NodeList`. Solche erhält man bspw. bei der Verwendung von DOM-Methoden wie `getElementsByClassName()` oder dem moderneren `querySelectorAll()`. Diese besitzen selbst keine Methoden wie `.map()` oder `.filter()`. Möchte man über eine solche also iterieren, muss man sie erst einmal in einen Array konvertieren. Dies geht mit ES2015 nun ganz einfach durch die Verwendung von `Array.from()`.

```
const links = Array.from(document.querySelectorAll('a'));
Array.isArray(links); // true
```

Methoden auf dem Array-Prototypen

Die Methoden auf dem Array-Prototypen können **direkt auf eine Array-Instanz** angewendet werden. Die gängigsten während der Arbeit mit React und insbesondere später mit Redux sind:

```
Array.find(func);
Array.findIndex(func);
Array.includes(value);
```

Die `Array.find()`-Methode dient, wie der Name es erahnen lässt, dazu, das **erste** Element eines Arrays zu finden, das bestimmte Kriterien erfüllt, die mittels der als ersten Parameter übergebenen Funktion geprüft werden.

```
const numbers = [1, 2, 5, 9, 13, 24, 27, 39, 50];
const biggerThan10 = numbers.find((number) => number > 10); // 13

const users = [
  { id: 1, name: 'Manuel' },
  { id: 2, name: 'Bianca' },
  { id: 3, name: 'Brian' },
];

const userWithId2 = users.find((user) => user.id === 2);
// { id: 2, name: 'Bianca'}
```

Die `Array.findIndex()`-Methode folgt der gleichen Signatur, liefert aber anders als die `Array.find()`-Methode nicht das gefundene Element selbst zurück, sondern nur dessen Index im Array. In den obigen Beispielen wären dies also `4` im ersten Beispiel, sowie `1` im zweiten.

Die in ES2016 neu dazu gekommene Methode `Array.includes()` prüft, ob ein Wert innerhalb eines Array existiert und gibt uns **endlich** einen Boolean zurück. Wer selbiges in der Vergangenheit mal mit `Array.indexOf()` realisiert hat, wird sich erinnern wie umständlich es war. Nun also ein simples `Array.includes()`:

```
[1, 2, 3, 4, 5].includes(4); // true
[1, 2, 3, 4, 5].includes(6); // false
```

Aufgepasst: die Methode ist case-sensitive. `['a', 'b'].includes('A')` gibt also `false` zurück.

Objekte

Statische Objekt-Methoden

Natürlich haben auch Objekte eine Reihe neuer Methoden und anderer schöner Möglichkeiten spendiert bekommen. Die wichtigsten im Überblick:

```
Object.assign(target, source[, source[,...]]);
Object.entries(Object)
Object.keys(Object)
Object.values(Object)
Object.freeze(Object)
```

Wieder der Reihe nach. Die wohl nützlichste ist aus meiner Sicht `Object.assign()`. Damit ist es möglich, die Eigenschaften eines Objekts oder auch mehrerer Objekte zu einem bestehenden Objekt hinzuzufügen (sozusagen ein Merge). Die Methode gibt dabei das Ergebnis als Objekt zurück. Allerdings findet dabei auch eine Mutation des **Ziel-Objekts** statt, weswegen die Methode mit Bedacht benutzt werden sollte. Beispiele sagen mehr als Worte, bitteschön:

```
const user = { id: 1, name: 'Manuel' };
const modifiedUser = Object.assign(user, { role: 'Admin' });
console.log(user);
// -> { id: 1, name: 'Manuel', role: 'Admin' }
console.log(modifiedUser);
// -> { id: 1, name: 'Manuel', role: 'Admin' }
console.log(user === modifiedUser);
// -> true
```

Hier fügen wir also die Eigenschaft `role` aus dem Objekt im zweiten Parameter der `Object.assign()`-Methode zum bestehenden **Ziel-Objekt** hinzu.

JSX – eine Einführung

JSX als wichtiger Bestandteil in der React-Entwicklung

Bevor wir tiefer in die Entwicklung von Komponenten einsteigen möchte ich zuerst einmal auf **JSX** eingehen, da JSX einen wesentlichen Teil bei der Arbeit mit React darstellt. Wie eingangs schon erwähnt stellt JSX einen ganz grundlegenden Teil der meisten React-Komponenten dar und ist aus meiner Sicht einer der Gründe, warum React so schnell und positiv von so vielen Entwicklern angenommen wurde. Mittlerweile bieten auch andere Frameworks wie Vue.js die Möglichkeit JSX zur komponentengetriebenen Entwicklung einzusetzen.

JSX sieht auf den ersten Blick erst einmal gar nicht sehr viel anders aus als HTML, oder eher noch XML, da in JSX, eben wie auch in XML und XHTML jedes geöffnete Element ein schließendes Element (`</div>`) besitzen oder selbstschließend (`<img />`) sein muss. Mit dem grundlegenden Unterschied, dass JSX auf **JavaScript-Ausdrücke** zurückgreifen kann und dadurch sehr mächtig wird.

Unter der Haube werden in JSX verwendete Elemente in einem späteren Build-Prozess in verschachtelte `React.createElement()`-Aufrufe umgewandelt. Wir erinnern uns zurück an die Einleitung. Dort hatte ich bereits kurz erwähnt, dass React eine Baumstruktur an Elementen erzeugt, die selbst aus verschachtelten `React.createElement()`-Aufrufen besteht.

Klingt jetzt alles fürchterlich kompliziert, ist aber ganz einfach. Sehen wir uns als Beispiel einmal das folgende kurze HTML-Snippet an:

```
<div id="app">
  <p>Ein Paragraph in JSX</p>
  <p>Ein weiterer Paragraph</p>
</div>
```

Wird dieses HTML so in dieser Form in **JSX** verwendet, werden diese Elemente später durch Babel in das folgende ausführbare JavaScript transpiliert:

```
React.createElement(
  'div',
  { id: 'app' },
  React.createElement('p', null, 'Ein Paragraph in JSX'),
  React.createElement('p', null, 'Ein weiterer Paragraph')
);
```

Das erste Funktionsargument für `createElement()` ist dabei jeweils der Tag-Name eines DOM-Elements als String-Repräsentation oder ein anderes JSX-Element, dann allerdings als Funktionsreferenz.

Das zweite Argument repräsentiert die **Props** eines Elements, also in etwa vergleichbar mit HTML-Attributen, wobei die Props in React deutlich flexibler sind und anders als herkömmliche HTML-Attribute nicht auf Strings beschränkt sind, sondern auch Arrays, Objekte oder gar andere React-Komponenten als Wert enthalten können.

Alle weiteren Argumente stellen die Kind-Elemente (*„children"*) des Elements dar. Im obigen Beispiel hat unser `div` zwei Paragraphen (`<p>`) als Kind-Elemente, welche selbst keine eigenen Props haben (`null`) und lediglich einen Text-String (`Ein Paragraph in JSX` bzw `Ein weiterer Paragraph`) als Kind-Element besitzt.

Wem das jetzt zu kompliziert klingt den kann ich beruhigen: Das geht in der Praxis hinterher wie von selbst von der Hand. Fast so, als würde man HTML-Markup schreiben. Dennoch halte ich es für wichtig die Hintergründe zumindest einmal gelesen zu haben um spätere Beispiele besser nachvollziehen zu können.

Ausdrücke in JavaScript

Was bedeutet dies nun für unsere JavaScript-Ausdrücke, auf die wir ja nun auch in JSX zurückgreifen können?

Zuerst einmal ist ein Ausdruck in JavaScript, kurz gesagt, ein Stück Code, das am Ende einen „Wert" erzeugt bzw ein „Ergebnis" zur Folge hat. Vereinfacht gesagt: alles was man bei der Variablenzuweisung auf die **rechte** Seite des Gleich-Zeichens (=) schreiben kann.

```
1 + 5;
```

… ist ein solcher Ausdruck, dessen Wert `6` beträgt.

```
'Hal' + 'lo';
```

… ist ein anderer Ausdruck der die zwei Strings `Hal` und `lo` per **String Concatenation** zu einem Wert `Hallo` zusammenfügt.

Stattdessen könnten wir aber auch einfach gleich schreiben:

```
6
'Hallo'
[1,2,3,4]
{a: 1, b: 2, c: 3}
```

```
true
null
```

… da **JavaScript-Datentypen** allesamt auch als Ausdruck verwendet werden können.

Die ES2015 **Template String Syntax**, die Backticks (```) benutzt, ist ebenfalls ein Ausdruck. Klar, sind sie doch letztlich nichts anderes als ein String:

```
`Hallo ${name}`;
```

Was hingegen **kein** Ausdruck ist, ist:

```
if (active && visibility === 'visible') { … }
```

… da ich zum Beispiel auch nicht schreiben könnte:

```
const isVisible = if (active && visibility === 'visible') { … }
```

Das würde mir jeder JavaScript-Interpreter wegen ungültiger Syntax um die Ohren hauen.

Lasse ich das `if` hier jedoch weg, habe ich einen **Logical AND Operator**, der wiederum ein Ausdruck ist und einen Wert zum Ergebnis hat (in diesem Fall `true` oder `false`):

```
const isVisible = active && visibility === 'visible';
```

Ebenso ist der Ternary-Operator (? :) ein Ausdruck:

```
Bedingung ? wahr : unwahr;
```

Ausdrücke sind aber nicht auf boolsche Werte, Nummern und Strings beschränkt sondern können auch Objekte, Arrays, Funktionsaufrufe und sogar Arrow-Functions sein, die ebenfalls neu in ES2015 eingeführt werden und uns hier nicht das letzte Mal begegnet sein werden.

Beispiel für eine Arrow-Function:

```
(number) => {
  return number * 2;
};
```

All das wird später noch wichtig werden. Um dem Ganzen jetzt endgültig die Krone aufzusetzen, können Ausdrücke wiederum selbst JSX beinhalten und so kann man das Spiel endlos weiterführen. Uff.

Da das später noch wichtig wird, hier nochmal einige Beispiele für JSX, das gültige Ausdrücke beinhaltet:

Simple Mathematik

```
<span>5 + 1 = {5 + 1}</span>
```

Ternary Operator

```
<span>Heute ist {new Date().getDay() === 1 ? 'Montag' : 'nicht Montag'}</span>
```

Ternary Operator als Wert einer Prop

```
<div className={user.isAdmin ? 'is-admin' : null}>…</div>
```

Array.map() mit JSX als Rückgabewert, das wiederum einen Ausdruck enthält

```
<ul>
  {['Tim', 'Struppi'].map((name) => (
    <li>{name})</li>
  ))}
</ul>
```

Zahlenwerte in Props

```
<input type="range" min={0} max={100} />
```

All dies sind erste Beispiele dessen, wie Ausdrücke dafür verwendet werden können, aus JSX mehr als nur simples HTML zu machen.

Was man außerdem wissen sollte

Wer die Beispiele aufmerksam studiert hat, dem werden je nach JavaScript-Kenntnissen vielleicht einige Dinge aufgefallen sein. Zuerst einmal tauchen in den Beispielen scheinbar willkürlich Klammern auf. Dies hat den Hintergrund, dass JSX stets in Klammern, also „(“ und „)“ geschrieben werden muss, wenn sich das JSX sich über mehr als eine Zeile erstreckt (also doch nicht willkürlich).

Prinzipiell schadet es nicht sein komplettes JSX immer in Klammern zu schreiben, auch wenn es sich nur um eine einzige Zeile handelt. Viele Leute bevorzugen das sogar aus Gründen der Einheitlichkeit, wirklich zwingend notwendig ist das aber nur bei mehrzeiligem JSX.

Möchten wir statt eines **Strings** einen **Ausdruck** innerhalb der Props nutzen wie im Beispiel *„Ternary Operator als Wert einer Prop“*, so nutzen wir dafür statt einfacher oder doppelter

Komponenten in React

Die zwei Erscheinungsformen von React Components

Eine erste einfache `HelloWorld`-Komponente haben wir schon beim Sprung ins kalte Wasser implementiert. Jedoch war dies natürlich bewusst eine sehr simple Komponente, die nicht gerade sehr praxisnah war und auch längst nicht alles beinhaltet hat was uns React bietet. Aber sie diente als gute erste Veranschaulichung, um die grundsätzliche Funktionsweise von von **React** und **React-Komponenten** kennenzulernen.

Das Prinzip von **Komponenten** ist einfach erklärt: eine **Komponente** erlaubt es komplexe User Interfaces in einzelne kleine Stücke zu unterteilen. Diese sind im Idealfall wiederverwendbar, isoliert und in sich geschlossen. Sie verarbeiten beliebigen Input von außen in Form sogenannter **Props** (engl. für „Properties", also Eigenschaften) und beschreiben letztendlich anhand ihrer `render()`-Funktion was auf dem Bildschirm erscheint.

Komponenten können grob in zwei verschiedenen Varianten auftreten: in Form einer einfachen Funktion (engl. **Function Component**), sowie **Class Components**, die eine gewöhnliche ES2015-Klasse repräsentieren.
Bis zur Einführung der React **Hooks** war es in **Function Components** nicht möglich, einen lokalen State zu verwalten, daher stößt man mancherorts noch auf den Begriff **Stateless Functional Component**, der aus einer Zeit stammt, als **Function Components** noch keinen State halten konnten. Wie **Hooks** im Detail funktionieren und wie State seit React 16.8.0 nun auch in **Function Components** verwendet werden kann, wird im weiteren Verlauf noch ausführlich im Kapitel über **Hooks** erläutert.

Function Components

Die deutlich einfachste Art in React eine Komponente zu definieren, ist sicherlich die **Function Component**, die, wie der Name es bereits andeutet, tatsächlich lediglich eine gewöhnliche JavaScript-Funktion ist:

```
function Hello(props) {
  return <div>Hello {props.name}</div>;
}
```

Diese Funktion erfüllt alle Kriterien einer gültigen **React-Komponente**: Sie hat als `return`-Wert ein explizites `null` (`undefined` ist dagegen **nicht** gültig!) oder ein gültiges `React.Element`

(hier in Form von **JSX**) und sie empfängt ein `props`-Objekt als erstes Funktionsargument, wobei sogar dieses optional ist und ebenfalls `null` sein kann.

Class Components / Klassen-Komponenten

Die zweite Möglichkeit wie eine **React-Komponente** erstellt werden kann habe ich im Eingangsbeispiel schon kurz gezeigt: **Class Components** oder auf gut Deutsch **Klassen-Komponenten**. Diese bestehen aus einer ES2015-Klasse, die von der `React.Component` oder `React.PureComponent`(dazu später mehr) Klasse ableitet und haben mindestens eine Methode mit dem Namen `render()`:

```
class Hello extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}
```

Wichtiger Unterschied hier: Während eine **Function Component** ihre **Props** als Funktionsargumente übergeben bekommt, bekommt die `render()`-Methode einer **Klassen-Komponente** selbst keinerlei Argumente übergeben, sondern es kann einzig über die Instanz-Eigenschaft `this.props` auf die **Props** zugegriffen werden!

Die beiden obigen Komponenten resultieren hier in einer komplett identischen Ausgabe!

> ⓘ Ein Kriterium, das beide Arten von Komponenten gemeinsam haben, ist, dass der `displayName`, also der Name einer gültigen Komponente, stets mit einem **Großbuchstaben** anfängt. Der Rest des Namens kann aus Großbuchstaben oder Kleinbuchstaben bestehen, wichtig ist lediglich, dass der erste Buchstabe stets ein Großbuchstabe ist!
>
> Beginnt der Name einer Komponente mit einem Kleinbuchstaben, behandelt React diese stattdessen als reines DOM-Element. `section` würde React also als DOM-Element interpretieren, während eine eigene Komponente durchaus den Namen `Section` haben kann und wegen ihres Großbuchstabens am Anfang von React korrekt vom `section` DOM-Element unterschieden werden würde.

Wie wir innerhalb der **Komponenten** jeweils mit dem **State** arbeiten, diesen modifizieren und uns zu Eigen machen, ist sehr komplex, weswegen dem Thema ein eigenes Kapitel gewidmet ist. Dieses folgt direkt im Anschluss an dieses hier und ich würde empfehlen, erst dieses Kapitel zu beenden um die Funktionsweise von Komponenten zu verstehen, bevor wir hier tiefer einsteigen.

State und Lifecycle-Methods

Kommen wir zu dem, was die Arbeit mit React erstmal wirklich effizient macht: **State** und die sogenannten **Lifecycle-Methods**.

Wie im vorangegangenen Kapitel bereits angesprochen, können Komponenten einen eigenen Zustand, den **State**, halten, verwalten und verändern. Dabei gilt der Grundsatz: **Ändert sich der State einer Komponente, löst dies immer auch ein Rerendering der Komponente aus!** Dieses Verhalten kann durch die Verwendung von `PureComponent` auch explizit unterbunden werden, was in einigen Fällen sinnvoll ist. Aber der Grundsatz bleibt unverändert: Eine State-Änderung löst ein Rerendering einer Komponente und ihrer Kind-Komponenten aus, außer diese sind selbst wiederum als `PureComponent` implementiert oder von einem `React.memo()`-Aufruf umschlossen.

Das ist insofern hilfreich, als wir nicht mehr manuell `ReactDOM.render()` aufrufen müssen, wann immer wir meinen, dass sich etwas an unserem Interface geändert hat, sondern die Komponenten dies stattdessen selbst entscheiden können.

Neben dem State an sich gibt es auch eine Handvoll sogenannter **Lifecycle-Methoden**. Dies sind Methoden, die **optional** in einer **Klassen-Komponente** definiert werden können und von React bei bestimmten Anlässen aufgerufen werden. Beispielsweise wenn eine Komponente erstmals gemountet wird, die Komponente neue Props empfangen oder sich der State innerhalb der Komponente geändert hat.

Erstmals seit **React 16.8.0** können mit der Einführung von **Hooks** auch **Function Components** einen eigenen **State** verwalten und auf bestimmte **Lifecycle**-Events reagieren. Dieses Kapitel handelt primär von Klassen-Komponenten, in denen State und Lifecycle-Events als Methoden der Klassen-Komponenten implementiert werden. Da **Hooks** ein sehr umfassendes und neues Konzept ist, habe ich ihnen ein eigenes Kapitel gewidmet, in dem uns die meisten Themen, die in diesem Kapitel behandelt werden noch einmal speziell für **Function Components** beschrieben erneut begegnen werden.

Eine erste stateful Component

Der **State** innerhalb einer Klassen-Komponente ist verfügbar über die Instanz-Eigenschaft `this.state` und ist somit innerhalb einer **Komponente** gekapselt. Weder Eltern- noch Kind-Komponenten können ohne Weiteres auf den State einer anderen Komponente zugreifen.

Um in einer Komponente einen initialen Zustand zu definieren gibt es zwei einfache Wege; einen dritten, von der Funktionalität her etwas erweiterten Weg, lernen wir später noch mit der

Lifecycle-Methode `getDerivedStateFromProps()` kennen.

Initialer State kann definiert werden, indem die Instanz-Eigenschaft `this.state` gesetzt wird. Dies ist entweder im Constructor einer **Klassen-Komponente** möglich:

```
class MyComponent extends React.Component {
  constructor(props) {
    this.state = {
      counter: props.counter,
    };
  }
  render() {
    // ...
  }
}
```

… oder indem der State als **ES2017 Class Property** definiert wird, was deutlich kürzer ist, jedoch momentan noch das **Babel-Plugin** `@babel/plugin-proposal-class-properties` (vor Babel 7: `babel-plugin-transform-class-properties`) benötigt:

```
class MyComponent extends React.Component {
  state = {
    counter: this.props.counter,
  };
  render() {
    // ...
  }
}
```

Create React App unterstützt die **Class Property Syntax** standardmäßig und da viele React-Projekte heute vollständig oder zumindest zu gewissen Teilen auf dem CRA-Setup oder Varianten davon basieren, kommt diese Syntax heute in den meisten Projekten zum Einsatz und kann genutzt werden. Sollte dies einmal in einem Projekt nicht der Fall sein, empfehle ich dringend die Installation und Nutzung des Babel-Plugins, da es wirklich viele unnötige Zeilen Code bei der täglichen Arbeit mit React einspart, während es gleichzeitig in nur wenigen Minuten eingerichtet ist.

Ist der **State** erst einmal definiert, können wir innerhalb der **Klassen-Komponente** mittels `this.state` **lesend** auf ihn zugreifen. **Lesend** ist hier ein entscheidendes Stichwort. Denn auch wenn es prinzipiell möglich, ist den State direkt über `this.state` zu verändern, sollte dies aus verschiedenen Gründen vermieden werden.

Den State verändern mit this.setState()

Um State zu verändern, stellt React eine eigene Methode innerhalb einer **Klassen-Komponente** bereit:

```
this.setState(updatedState);
```

Wann immer der State innerhalb einer Komponente verändert werden soll, sollte dafür `this.setState()` verwendet werden. Der Aufruf von `this.setState()` führt dann dazu, dass React entsprechende **Lifecycle-Methoden** (wie bspw. `componentDidUpdate()`) ausführt und eine Komponente **neu rendert!** Würden wir den State stattdessen direkt verändern, also bspw. `this.state.counter = 1;` schreiben, hätte dies vorerst keinerlei Auswirkungen auf unsere Komponente und alles würde aussehen wie bisher, da der Render-Prozess **nicht** ausgelöst werden würde. React hat dann keine Kenntnis von der Änderung am State!

Die Methode ist von der Funktionsweise her allerdings etwas komplexer, als es auf den ersten Moment aussehen mag. Und so wird nicht einfach nur der alte State durch den neuen State ersetzt und ein Rerendering ausgelöst; es passieren auch noch allerhand andere Dinge. Der Reihe nach.

Zuerst einmal kann die Funktion **zwei verschiedene Arten von Argumenten** entgegennehmen. Das ist einerseits ein **Objekt** mit neuen oder aktualisierten State-Eigenschaften, sowie andererseits eine **Updater-Funktion,** die wiederum ein Objekt zurückgibt oder `null`, falls nichts geändert werden soll. Bestehende gleichnamige Eigenschaften innerhalb des State-Objekts werden dabei **überschrieben**, alle anderen bleiben **unangetastet!** Möchten wir eine Eigenschaft im State zurücksetzen, müssen wir diese dazu also explizit auf `null` oder `undefined` setzen. Der übergebene State wird also immer mit dem bestehenden State **zusammengefügt**, niemals **ersetzt!**

Nehmen wir nochmal unseren oben definierten State mit einer `counter` Eigenschaft, deren initialer Wert für dieses Beispiel erst einmal `0` ist. Nun verändern wir den State und möchten diesem zusätzlich eine `date`-Eigenschaft mit dem aktuellen Datum hinzufügen. Übergeben als Objekt wäre unser Aufruf:

```
this.setState({
  date: new Date(),
});
```

Nutzen wir stattdessen eine **Updater-Funktion**, wäre unser Aufruf:

```
this.setState(() => {
  return {
    date: new Date(),
```

CSS und Styling

Styling in React ist ein relativ eigenes Thema. So bietet React keine Hausmittel an, um das Styling von Anwendungen mittels CSS zu erleichtern, mit CSS-in-JS hat sich jedoch eine ganz eigene, teilweise sehr kontrovers diskutierte Bewegung zu der Thematik gebildet. Dabei wird der Styling-Teil ebenfalls in JavaScript umgesetzt, um nicht mit dem Paradigma der komponentenbasierten Entwicklung zu brechen. Doch eins nach dem anderen, starten wir erst einmal mit den Basics und arbeiten uns dann Stück für Stück an das Thema heran.

Styling mittels style-Attribut

Die wahrscheinlich einfachste Möglichkeit um in React Komponenten zu stylen, ist durch die Verwendung des `style`-Attributs auf HTML-Elementen. Dieses funktioniert jedoch etwas anders als in HTML und so erwartet React ein **Objekt** in der Form `Eigenschaft: Wert`, wobei als Eigenschaft die JavaScript-Schreibweise einer CSS-Eigenschaft erwartet wird. Also etwa `zIndex` statt `z-index`, `backgroundColor` statt `background-color` oder `marginTop` statt `margin-top`. Bei Werten, die Pixel-Angaben akzeptieren, ist die Verwendung von `px` als Einheit optional:

```
<div style={{ border: '1px solid #ccc', marginBottom: 10 }}>
  Ein div mit einem grauen Rahmen und 10 Pixel Abstand nach unten
</div>
```

Werte, die ohne Einheit verwendet werden (bspw. `z-index`, `flex` oder `fontWeight`) sind davon nicht betroffen, sie können bedenkenlos benutzt werden, ohne dass React sie um `px` ergänzt.

Durch die Verwendung eines Objekts anstelle eines Strings ist React konsistent zur `style`-Eigenschaft von DOM-Elementen (`document.getElementById('root').style` ist ebenfalls ein Objekt!) und sorgt außerdem dafür, dass keine Sicherheitslücken durch XSS entstehen können.

Nun ist die Verwendung von Inline-Styles nicht unbedingt wünschenswert, sie kann aber in manchen Situationen hilfreich sein, etwa wenn das Styling eines Elements von bestimmten variablen Werten im State abhängt.

Verwendung von CSS-Klassen in JSX

Wesentlich angenehmer und sauberer ist die Verwendung von echten CSS-Klassen in JSX, wie es eben auch schon aus der Verwendung on HTML bekannt ist. Der entscheidende Unterschied

hier: anders als in HTML lautet der Name der entsprechenden Prop in JSX nicht `class`, sondern `className`:

```
<div className="item">...</div>
```

React rendert hier schließlich gewohnte Syntax mit der jeweiligen HTML-Entsprechung:

```
<div class="item">...</div>
```

Wie in JSX üblich kann, der Wert für die `className`-Prop auch dynamisch über String Concatenation zusammengesetzt werden:

```
render() {
  let className = 'item';
  if (this.state.selectedItem === this.props.itemId) {
    className += ' item--selected';
  }
  return (
    <div className={className}>
      ...
    </div>
  );
}
```

Hier wäre der Wert für `className` in jedem Fall `item` und für den Fall, dass das ausgewählte Item dem aktuellen Item entspricht `item item--selected`.

Als de facto Standard für derartige Fälle hat sich das Paket `classnames` etabliert. Mit diesem ist es möglich, Klassen abhängig von einer Bedingung zu machen. Installiert werden kann es über die CLI, mittels:

```
npm install classnames
```

```
yarn add classnames
```

Anschließend müssen wir es nur noch in den Komponenten importieren, in denen wir es nutzen wollen. Dies passiert ganz unkompliziert via:

```
import classNames from 'classnames';
```

Damit importieren wir die Funktion und weisen ihr den Namen `classNames` zu. Die Funktion erwartet dann beliebig viele Parameter die entweder ein String sein können oder ein Objekt in der Form `{ Klasse: true|false }`. Die Funktionsweise ist dabei ebenfalls recht simpel: Ist

der Wert für eine Eigenschaft `true`, setzt `classNames` eine Klasse mit dem selben Eigenschaftsnamen. Im obigen Beispiel wäre das also:

```
render() {
  return (
    <div className={classNames('item', {
      'item--selected': this.state.selectedId === this.props.itemId
    })}>
      ...
    </div>
  );
}
```

Auch die Verwendung eines Objekts mit mehreren Eigenschaften ist möglich, hier werden dann sämtliche Eigenschaften als Klasse gesetzt, deren Wert `true` ist:

```
render() {
  return (
    <div className={classNames({
      'item': true,
      'item--selected': this.state.selectedId === this.props.itemId
    })}>
      ...
    </div>
  );
}
```

Bei der Verwendung von Klassen sollte natürlich sichergestellt sein, dass das Stylesheet mit den jeweiligen Klassen auch im HTML-Dokument entsprechend eingebunden wird, da sich React darum prinzipiell nicht kümmert.

Modulares CSS mit CSS Modules

CSS Modules sind eine Art Vorstufe zu **CSS-in-JS** und vereinen einige Eigenschaften von CSS und JavaScript-Modulen in sich. Wie der Name es suggeriert, befindet sich das CSS hier in eigenen importierbaren **Modulen**, die jedoch aus reinem CSS bestehen und zumeist unmittelbar zu einer Komponente gehören. Entwickeln wir etwa eine Komponente um ein Profilbild darzustellen in einer Datei mit dem Namen `ProfileImage.js`, existiert beim **CSS Modules** Ansatz oft auch eine Datei mit dem Namen `ProfileImage.module.css`. Beim Import eines solchen CSS Moduls wird dann sichergestellt, dass ein einmaliger, oftmals etwas kryptischer Klassenname generiert wird, um so sicherzustellen, dass diese Klasse nur in der jeweiligen Komponente verwendet wird. Konflikte mit anderen Komponenten, die gleiche Klassennamen verwenden, sollen so ausgeschlossen werden.

Dabei sind **CSS Modules** selbst jedoch erstmal nur ein **Konzept** und noch keine konkrete Implementierung. Die Implementierung übernehmen dann Tools wie bspw. `css-loader` für Webpack.

Die Endung `.module.css` ist dabei keineswegs zwingend und auch ein bloßes `.css` wäre möglich, `.module.css` hat sich aber bei einigen bekannten Tools mit großer Reichweite als Konvention entwickelt und wird auch von **Create React App** ohne die Notwendigkeit des Vornehmens weiterer Konfigurationseinstellungen unterstützt. CRA greift dabei auf die eben angesprochene `css-loader`-Implementierung für Webpack zurück.

In den CSS-Dateien befindet sich reguläres, **standardkonformes CSS** in seiner gewohnten Schreibweise. Dieses kann dann in einer Komponente wie ein JavaScript-Modul importiert werden:

```
import css from './ProfileImage.module.css';
```

Hier kommt dann die Besonderheit von **CSS Modules** zum Tragen: in der Variable `css` befindet sich nun ein Objekt mit Eigenschaften, die den Klassennamen aus unserem CSS Modul entsprechen. Existiert in der CSS-Datei eine CSS-Klasse mit dem Namen `image` haben wir nun im `css`-Objekt ebenfalls eine Eigenschaft mit dem Namen `image`. Dabei ist der Wert ein einmaliger, generierter String wie z.B. `ProfileImage_image_2cvf73`.

Diesen generierten Klassennamen können wir nun in unserer Komponente als `className` vergeben:

```
<img src={props.imageUrl} className={css.image} />
```

Und das Resultat beim Rendering wäre:

```
<img src="..." className="ProfileImage_image_2cvf73" />
```

Nutzen wir nun die Klasse `image` noch in einer anderen Komponente und importieren dort ebenfalls eine CSS-Datei mit einer `image`-Klasse würde es hier, anders als in gewöhnlichem CSS, **nicht** zu einem Konflikt kommen, da der *generierte* Klassenname ein anderer wäre.

Dabei ist alles erlaubt, was auch in CSS erlaubt ist und auch die Kaskade (also das **C** in **Cascading Style Sheets**) bleibt dabei intakt:

```
.imageWrapper img {
  border: 1px solid red;
}

.imageWrapper:hover img {
```

```
  border-color: green;
}
```

Das obige CSS würde bei folgender JSX-Struktur auf das Bild angewendet werden:

```
const ProfileImage = () => {
  return (
    <div className={css.imageWrapper}>
      <img src="..." />
    </div>
  );
};
```

Der Hintergrund für **CSS Modules** ist eben der, dass mögliche Konflikte bei der Entwicklung von sehr isolierten Komponenten eben ausgeschlossen werden sollen, während das eigentliche Styling selbst weiterhin in CSS-Dateien und ganz ohne JavaScript-Kenntnisse vorgenommen werden kann. Ein idealer Kompromiss insbesondere in Teams, in denen die Entwicklung von JavaScript und CSS bislang recht strikt getrennt wurde und es Experten für beide Disziplinen im Team gibt.

CSS-in-JS – die Verlagerung von Styles ins JavaScript

In der Einleitung hatte ich es bereits angesprochen: **CSS-in-JS** wird in einigen Entwicklerkreisen mitunter sehr rege und kontrovers diskutiert. Gegner führen das Argument ins Feld, dass die Befürworter von **CSS-in-JS** einfach nicht verstanden hätten, die Kaskade zu verwenden, um skalierbares CSS zu schreiben. Befürworter argumentieren dagegen meist, dass es nicht um die Kaskade geht, sondern man ganz einfach einen sicheren Weg benötige, um höchstmögliche Isolation von Komponenten sicherzustellen. Nun, ich persönlich bin da eher diplomatisch und denke, dass es für beide Seiten Argumente gibt, doch vor allem glaube ich, dass **CSS-in-JS** durchaus seine Daseinsberechtigung hat. Doch worum geht es überhaupt genau?

Beim **CSS-in-JS**-Ansatz wird der Styling-Teil, der in der klassischen Web-Entwicklung immer die Aufgabe von CSS war, in den JavaScript-Teil verschoben. Wie auch bei der **CSS Modules** „Hybrid"-Lösung steht hier vor allem die Isolation und konfliktfreie Wiederverwendbarkeit von JavaScript-Komponenten im Vordergrund. Anders als bei **CSS Modules** findet das Styling in **CSS-in-JS** jedoch vollständig in JavaScript-Dateien statt, allerdings noch immer unter Verwendung von CSS oder zumindest CSS-naher Syntax.

CSS-in-JS ist dabei ebenfalls nur ein Oberbegriff für das Konzept als solches, die konkrete Implementierung findet dann durch die Verwendung von einer von mittlerweile sehr vielen Libraries statt. Eine Übersicht über die zur Auswahl stehenden Optionen bietet diese Liste:

http://michelebertoli.github.io/css-in-js/[23] – hier werden insgesamt über 60 verschiedene Libraries gelistet, die als **CSS-in-JS** Implementierung benutzt werden können.

In diesem Kapitel wollen wir uns auf **Styled Components** als kurzes Beispiel beschränken, das mit über 23.000 Stars auf GitHub die populärste Option darstellt. Dazu müssen wir **Styled Components** zunächst als Abhängigkeit in unser Projekt über die Kommandozeile installieren:

```
npm install styled-components
```

```
yarn add styled-components
```

Ist das Paket erst einmal installiert, können wir es importieren und dann gleich mit einer ersten **Styled Component** beginnen. Dazu importieren wir die `styled`-Funktion und nutzen das jeweilige HTML-Element, das wir stylen wollen, als Eigenschaft. Anschließend nutzen wir die **Template Literal** Syntax aus ES2015 um das CSS für unsere **Styled Component** zu definieren.

Um einen schwarz-gelben Button zu erzeugen, sähe das in einem vollständigen Beispiel etwa so aus:

```
import React from 'react';
import ReactDOM from 'react-dom';
import styled from 'styled-components';

const Button = styled.button`
  background: yellow;
  border: 2px solid black;
  color: black;
  padding: 8px;
`;

const App = () => {
  return (
    <div>
      <h1>Beispiel für eine Styled Component</h1>
      <Button>Ein schwarz-gelber Button ohne Funktion</Button>
    </div>
  );
};

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

Durch die Verwendung von `const Button = styled.button` erzeugt **Styled Components** dabei eine neue `Button`-Komponente und weist der Komponente die CSS-Eigenschaften zu, die

wir im darauffolgenden **Template Literal** definieren. Innerhalb dieses **Template Literals** wird dann wiederum herkömmliches CSS verwendet.

Möchten wir Pseudo-Selektoren oder -Elemente verwenden, nutzen wir diese einfach ohne vorangestellten Selektor. Im obigen Beispiel sieht das für einen `:hover`-Status etwa so aus:

```
const Button = styled.button`
  background: yellow;
  border: 2px solid black;
  color: black;
  padding: 8px;
  :hover {
    background: gold;
  }
`;
```

Auch der Zugriff auf die **Props** einer **Styled Component** ist möglich. Dazu wird im Template String eine Funktion aufgerufen, die als ersten Parameter alle **Props** des jeweiligen Elements übergeben bekommt:

```
const Button = styled.button`
  background: yellow;
  border: 2px solid black;
  color: black;
  cursor: ${(props) => (props.disabled ? 'not-allowed' : 'pointer')};
  padding: 8px;
  :hover {
    background: gold;
  }
`;
```

Hier würde etwa der Mauszeiger zu einem `not-allowed` Symbol werden, wenn der Button eine `disabled`-Eigenschaft besitzt. Darüber hinaus bietet Styled Components auch Unterstützung für Themes, serverseitiges Rendering, CSS-Animationen und vieles mehr. Für eine (sehr) detaillierte Übersicht empfehle ich an dieser Stelle, einen Blick auf die gute und umfassende Dokumentation[24] zu werfen.

Die positiven Seiten von **CSS-inJS** generell und **Styled Components** im Speziellen liegen hier auf der Hand und die Dokumentation fasst sie ziemlich gut zusammen:

- kritisches CSS, also das, was für die aktuell aufgerufene Seite relevant ist, wird automatisch generiert, da **Styled Components** genau weiß, welche Komponenten auf einer Seite verwendet werden und welche Styles diese benötigen

- durch die automatische Generierung von Klassennamen wird das Risiko von Konflikten auf ein Minimum reduziert
- dadurch, dass CSS unmittelbar an eine Komponente gebunden wird, ist es sehr einfach, nicht mehr benutztes CSS zu finden. Wird die erzeugte **Styled Component** nicht mehr in einer Anwendung verwendet, wird auch das CSS nicht mehr benötigt und die Komponente kann mitsamt ihres CSS sicher gelöscht werden
- Komponenten-Logik (JavaScript) und Komponenten-Styling (CSS-in-JS) befinden sich unmittelbar am selben Ort, teilweise sogar in der gleichen Datei. Der Entwickler muss nicht erst mühsam die Stelle finden, die er ändern muss bei einer Style-Änderung
- **Styled Components** erzeugt ohne jede weitere Konfiguration automatisch CSS, das Vendor-Prefixes für alle Browser enthält

Nun ist **Styled Components** nur eine von wie bereits angesprochen sehr vielen Implementierungen des Konzepts **CSS-in-JS**. Sie ist ohne Zweifel die bekannteste und am weitesten adaptierte und auch in einigen meiner vergangenen Projekte hat sie ihren Job tadellos erfüllt. Dennoch schlage ich vor, vielleicht einen Blick auf andere, verfügbare Alternativen zu werfen, um die für den jeweiligen eigenen Verwendungszweck optimale Lösung zu finden. Zu den ebenfalls bekannteren, erwähnenswerten Beispielen gehören:

- emotion[25]
- styled-jsx[26]
- react-jss[27]
- radium[28]
- linaria[29]

III – Erweiterte Konzepte

Higher Order Components

Higher Order Components (meist abgekürzt: **HOC** oder **HOCs**) waren und sind ein sehr zentrales Konzept bei der Arbeit mit React. Sie erlauben es, Komponenten mit wiederverwendbarer Logik zu implementieren und sind angelehnt an **Higher Order Functions** aus der funktionalen Programmierung. Das Prinzip hinter derartigen Funktionen ist, dass sie eine Funktion als Parameter entgegennehmen und eine neue Funktion zurückgeben. Im Fall von React wird das Prinzip auf Komponenten angewandt. Daher der von den **Higher Order *Functions*** abgeleitete Name **Higher Order *Component***.

Zum leichteren Verständnis gleich ein erstes einfaches Beispiel:

```
const withFormatting = (WrappedComponent) => {
  return class extends React.Component {
    bold = (string) => {
      return <strong>{string}</strong>;
    };
    italic = (string) => {
      return <em>{string}</em>;
    };
    render() {
      return <WrappedComponent bold={this.bold} italic={this.italic} />;
    }
  };
};
```

Hier haben wir eine Funktion `withFormatting` definiert, die eine React-Komponente entgegen nimmt. Die Funktion gibt dabei eine neue React-Komponente zurück, welche die *in die Funktion herein gegebene Komponente* rendert und ihr dabei die Props `bold` und `italic` übergibt. Die hereingegebene Komponente kann nun auf diese Props zugreifen:

```
const FormattedComponent = withFormatting(({ bold, italic }) => (
  <div>
    Dieser Text ist {bold('fett')} und {italic('kursiv')}.
  </div>
));
```

Typischerweise werden **Higher Order Components** benutzt um Logik darin zu kapseln. In diesem Zusammenhang ist auch oft die Rede von **Smart** und **Dumb Components** also: **schlaue** und **dumme** Komponenten. Die Smart Components (zu denen Higher Order Components zählen) sind dann dazu da, Business Logik abzubilden, mit APIs zu kommunizieren oder

Verhaltenslogik zu verarbeiten. *Dumme* Komponenten hingegen bekommen weitestgehend statische Props übergeben und beschränken den Logik-Teil auf reine Darstellungslogik. Also bspw. ob ein Benutzerbild, oder falls dieses nicht vorhanden ist, stattdessen ein Platzhalterbild angezeigt wird. In diesem Zusammenhang fällt auch oft der Begriff **Container Component** (für *Smart* Components) und **Layout Components** (für *Dumb* Components).

Doch wozu das überhaupt? Eine solche strikte Unterteilung in Business Logik und Darstellungslogik macht echte komponentenbasierte Entwicklung erst einmal möglich. Sie erlaubt es, Layout-Komponenten zu erstellen die keinerlei Kenntnis von etwaigen APIs haben und nur stumpf die Daten darstellen, die ihnen übergeben werden, völlig egal woher diese kommen. Gleichzeitig erlaubt sie es auch den Business Logik Komponenten, sich um die reine Business Logik zu kümmern, völlig gleichgültig wie die Daten letzten Endes dargestellt werden.

Stellen wir uns ein gängiges Beispiel aus der Interface Entwicklung einmal vor: die Umschaltung zwischen einer **Listen-** und einer **Karten-Ansicht**. Hier würde sich eine **Container-Komponente** darum kümmern, die Daten zu beschaffen die relevant für den Benutzer sind. Sie würde die beschafften Daten dann an die frei konfigurierbare **Layout-Komponente** übergeben. Solange beide Komponenten sich an das vom Entwickler vorgebene Interface (Stichwort **PropTypes**) halten, sind beide Komponenten beliebig austauschbar und können vollkommen unabhängig voneinander entwickelt und **getestet** werden!

Genug Theorie. Zeit für ein weiteres Beispiel. Wir wollen uns eine Liste mit den 10 größten Kryptowährungen laden und ihren momentanen Preis anzeigen. Dazu erstellen wir eine **Higher Order Component,** die sich diese Daten über die frei zugängliche Coinmarketcap API beschafft und an eine Layout-Komponente übergibt.

```
const withCryptoPrices = (WrappedComponent) => {
  return class extends React.Component {
    state = {
      isLoading: true,
      items: [],
    };

    componentDidMount() {
      this.loadData();
    }

    loadData = async () => {
      this.setState(() => ({
        isLoading: true,
      }));

      try {
        const cryptoTicker = await fetch(
```

```
        {this.state.modalIsOpen && (
          <ModalPortal>
            <p>Dieser Teil wird in einem Modal-Fenster geöffnet.</p>
            <button onClick={this.closeModal}>Modal schließen</button>
          </ModalPortal>
        )}
      </div>
    );
  }
}

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

Ein besonderes Augenmerk gilt hier der `this.closeModal` Methode. Diese wird als Methode der App-Komponente definiert, wird aber innerhalb der `ModalPortal`-Komponente beim Klick auf den „Modal schließen"-Button im Kontext der `App`-Komponente aufgerufen.

Sie kann also problemlos den `modalIsOpen` State der Komponente verändern. Und das, obwohl die Komponente sich gar nicht innerhalb `<div id="root>` befindet wie der Rest unserer kleinen App. Dies ist möglich, da es sich eben um ein Portal handelt, deren Inhalt sich **aus React-Sicht** im selben Komponenten-Baum wie die App selbst befindet, nicht jedoch **aus HTML-Sicht**.

Code Splitting

Wer ein Projekt mit React entwickelt, nutzt in den allermeisten Fällen auch einen **Bundler** wie **Webpack, Browserify** oder **Rollup**. Diese sorgen dafür, dass alle einzelnen Dateien, alle Imports, später zu einer einzigen großen Datei gebündelt wird, die dann relativ einfach deployed werden kann, ohne dass sich ein Entwickler noch all zu viele Gedanken um relative Verlinkungen machen muss. Dieser Vorgang wird dementsprechend **Bundling** genannt. In sehr großen und komplexen Projekten kann so ein **Bundle** schnell mal ein Megabyte groß oder gar größer werden, insbesondere wenn viele Third-Party-Bibliotheken im Einsatz sind. Das ist in vielerlei Hinsicht ein Problem, denn große Bundles benötigen länger, um vom Browser heruntergeladen zu werden und auch das Ausführen unnötig großer Bundles führt unweigerlich zu Performance-Einbußen.

Um dem Problem der großen Bundles zu begegnen, gibt es das sog. **Code Splitting**. Beim Code Splitting wird die Anwendung in mehrere kleinere Bundles aufgeteilt, die allesamt für sich allein gesehen lauffähig sind und weitere Bundles nachladen, sollten diese später benötigt werden. So ist die Aufteilung in ein Bundle mit den meist benutzten Abhängigkeiten (bspw. React, React DOM, …) und jeweils ein Bundle pro Route eine recht gängige Methode beim **Code Splitting**.

Die einfachste Methode dazu ist die Verwendung der **Dynamic Import Syntax**. Diese ist momentan ein Proposal beim **TC39**, befindet sich also momentan im Standardisierungsprozess. Dank Webpack und Babel ist es aber auch heute schon möglich, die Verwendung zu benutzen. Notwendig ist hierfür das Babel Plugin `@babel/plugin-syntax-dynamic-import`. **Create React App** und andere Tools wie **next.js** oder **Gatsby** bringen von Haus aus Unterstützung für dynamische Imports mit und müssen nicht speziell für die Verwendung von **Code Splitting** konfiguriert werden.

Verwendung dynamischer Imports

Im Kapitel zu ES2015+ wurde die Import-Syntax bereits angesprochen. Die **Dynamic Import Syntax** ist eine Erweiterung dieser und erlaubt es, daher der Name, innerhalb einer Anwendung Imports dynamisch nachzuladen. Dabei funktioniert ein dynamischer Import nicht anders als ein Promise:

```
// greeter.js
export sayHi = (name) => `Hi ${name}!`;
```

```
// app.js
import('./greeter').then((greeter) => {
```

IV – Hooks

Einführung in Hooks

Hooks sind ein komplett neues Konzept, das in React **16.8.0** eingeführt wurde. So komplett neu, dass ich **Hooks** in diesem Buch ein eigenes Kapitel widmen möchte. Einige der React Core-Entwickler haben es bereits als die aufregendste und grundlegendste Änderung an React in den letzten Jahren bezeichnet und in der Tat haben **Hooks** viel Aufsehen erregt, als sie auf der **React Conf 2018** angekündigt und gleich im Anschluss in der ersten Alpha-Version von React 16.7. eingeführt wurden. Inzwischen haben auch andere Frameworks eigene **Hooks** implementiert. Doch worum geht es eigentlich?

Mit **Hooks** ist es erstmals möglich, Mechanismen in **Function Components** zu benutzen die bisher nur in Klassen-Komponenten möglich waren. Features wie `setState` oder Lifecycle Methoden, ähnlich wie `componentDidMount()` oder `componentDidUpdate()`, werden dank **Hooks** auch in **Function Components** möglich. **Hooks** sind dabei im wesentlichen Sinn nichts anderes als spezielle Funktionen, die einem festen Schema folgen. So gibt es die Konvention, dass **Hooks** zwangsweise mit `use` anfangen müssen.

Dabei stellt React selbst eine Reihe interner **Hooks** wie etwa `useState`, `useEffect` oder `useContext` zur Verfügung, erlaubt aber auch die Erstellung eigener Hook-Funktionen, den sogenannten **Custom Hooks**, in denen dann eigene Logik gebündelt werden kann. Auch diese müssen per Konvention mit `use` beginnen, darüber hinaus ist die weitere Benennung dem Entwickler selbst überlassen, der Name muss lediglich ein valider JavaScript-Funktionsname sein. Also etwa `useAccountInfo` oder `useDocumentInfo`

> Kleine persönliche Anekdote am Rande: die Einführung von **Hooks** hat dazu geführt dass ich große Teile dieses Buches umformulieren musste und auch noch weiterhin umformulieren muss. So war bspw. vorher auch in der offiziellen Dokumentation die Rede von **Stateless Functional Components** (oder kurz **SFCs**), das **Stateless** wurde mit der Einführung von **Hooks** quasi über Nacht aus der offiziellen Dokumentation gestrichen, was ich zum Anlass nahm, dies auch in diesem Buch zu tun.

Hooks wurden vor dem Hintergrund eingeführt, dass es mit ihnen erstmals möglich sein soll, Komponenten-Logik in einer einheitlichen und von React vorgegebenen Form zwischen verschiedenen Komponenten zu teilen. Vor **Hooks** kam es häufig vor, dass verschiedene Komponenten fast identische `componentDidMount()` oder `componentDidUpdate()`-Methoden implementiert haben; oder, dass vor allem innerhalb einer Klassen-Komponente speziell diese beiden Methoden eine nahezu identische Implementierung hatten, mit dem einzigen

Unterschied, dass in `componentDidUpdate()` zusätzlich geschaut wurde, ob sich gewisse Parameter geändert haben. Bspw., ob sich eine via Props übergebene Benutzer-ID geändert hat, woraufhin ein API-Request initiiert wurde, um die Daten für den jeweiligen Benutzer abzufragen.

Aus diesem Grund wurde mit den **Hooks** ein neues Konzept eingeführt, mit dem komplexe Logik deutlich einfacher und ohne viel duplizierten Code geschrieben werden kann. Sie erfordern tatsächlich, wenn man mit der Funktionsweise von **Klassen-Komponenten** erst einmal vertraut ist, etwas Umdenken. Da sich einige Abläufe und der Aufbau der Komponenten selbst etwas ändern, hat man es schließlich nur noch mit relativ simplen Funktionen zu tun statt wie vorher mit komplexen Klassen mit Klassen-Methoden, Klassen-Eigenschaften, Vererbung und einem gemeinsamen `this`-Kontext. Doch zu den genauen Details kommen wir noch im weiteren Verlauf dieses Kapitels.

Sind Klassen-Komponenten jetzt schlecht?

Bleibt noch die Frage zu klären: *sind Klassen-Komponenten jetzt schlecht?*

Diese Frage kam unmittelbar nach der Ankündigung und Einführung von **Hooks** in der React-Community immer wieder auf. Das React-Team beantwortete es so, dass sie nicht empfehlen würden, eine bestehende Anwendung, die mit **Klassen-Komponenten** arbeitet, nicht abrupt in **Function Components** mit **Hooks** umzuschreiben, da auch **Klassen-Komponenten** weiterhin Teil von React bleiben werden.

Die Community hat das aber zu großen Teilen nicht wirklich interessiert und so gab es nach dem Release von **React 16.7.0-alpha.0**, also der ersten Version mit **Hooks**, zahlreiche Meldungen von Entwicklern auf Twitter, die sich trotz aller Warnungen nicht davon abhalten ließen, ihre Anwendungen mit **Hooks** umzuschreiben und zum deutlich überwiegenden Teil begeistert waren von der neuen Einfachheit der Entwicklung von Komponenten ohne den Overhead, den Klassen-Komponenten ein Stück weit mit sich gebracht haben.

Grundsätzlich ist also auch nach wie vor nichts gegen die Verwendung von Klassen-Komponenten einzuwenden und wer mag, kann diese auch weiterhin verwenden, da es momentan keine Pläne gibt, diese wieder aus React zu entfernen. Wer sich allerdings erst einmal an die Verwendung von **Hooks** gewöhnt hat, dem dürfte es in den meisten Fällen schwer fallen, freiwillig auf die Einfachheit und Verständlichkeit dieser neuen Form einer Komponente zu verzichten.

Klassen-Komponenten und Hooks – ein Vergleich

Um zu veranschaulichen, wie viel simpler Komponenten durch die Verwendung von **Hooks** werden können hat Sunil Pai, ebenfalls ein Core-Entwickler von React, einen Vergleich erstellt, bei dem zusammenhängende Logik gleich eingefärbt wurde und die Teile, die in der **Function**

Component mit **Hooks** nicht mehr benötigt werden, in der Klassen-Komponente geschwärzt wurden. Das Ergebnis ist ein sehr harmonisches Bild in dem Logik jeweils an einer Stelle gebündelt ist und nicht an verschiedenen Stellen innerhalb der Komponente verwendet wird:

V – Das Ecosystem

Für die einen Fluch, für die anderen Segen: die **Freiheit** die **React** dem Entwickler bietet.

Während vollwertige Frameworks wie Angular sehr klare Vorgaben darüber machen, wie eine Anwendung strukturiert werden soll und auch gleich eigene Methoden für Datenhaltung, Services und Business-Logik mitbringen sowie klare Wege vorgeben, funktioniert das in React eher nach dem Motto „bring your own“ – also bringe dein eigenes Werkzeug mit.

Wie bereits zu Beginn dieses Buches erwähnt, ist React erst einmal nur eine **Bibliothek zur Entwicklung von User Interfaces** (Kleine Anekdote am Rande: Dieser Satz ist einer der wenigen, die ich in diesem Buch aus der Doku zitiert habe, die sich im Laufe der Arbeit an diesem Buch nicht geändert hat ;)). In der Welt der klassischen MVC (*Model View Controller*) Architektur also sozusagen nur der View-Layer. Wer darüber hinaus ein erweitertes State-Management möchte, wer seine Anwendung mehrsprachig entwickeln möchte oder clientseitiges Routing benötigt, wird hier mit React allein oftmals nicht glücklich.

Doch hier hat sich ein sehr wertvolles und aktives Ecosystem um React herum entwickelt mit Tools, die sich hervorragend in ein React-Setup integrieren lassen und den Entwickler vor die Wahl stellen, sich aus mehreren Tools das jeweilige herauszusuchen, dass seinem persönlichen Geschmack am meisten zusagt. Passt gar keins, ist meist auch das kein Problem und React bietet mit seiner Vielzahl an Funktionen und APIs die Möglichkeit, sich eine robuste, eigene Lösung zu entwickeln.

Da es durch die Vielzahl an Libraries sehr schnell passieren kann, dass man den Überblick verliert, möchte ich in diesem Kapitel einmal auf die gängigsten Tools und Libraries eingehen, die sich in der täglichen Arbeit mit React bewährt haben und oftmals tausende von Stars auf Github haben und zehntausendfach oder (deutlich) mehr über `npm` installiert werden.

Routing

Eine Funktionalität, die in nahezu allen **Single Page Applikationen** (SPA) früher oder später (meist früher als später) benötigt wird, ist das **Routing**. Also das Zuordnen einer URL in der Anwendung zu einer bestimmten Funktion. Rufe ich bspw. die URL `/users/manuel` auf, möchte ich dort sehr wahrscheinlich das Benutzerprofil des Users `manuel` anzeigen.

Hier hat sich der **React Router** in den vergangenen Jahren als de facto Standard etabliert. Entwickelt von Michael Jackson (ja, der Kerl heißt wirklich so!) und Ryan Florence (der laut eigener Aussage inzwischen über 10 verschiedene Router für diverse Zwecke entwickelt hat) bringt er es auf mittlerweile über 35.000 Stars bei GitHub. Das Paket wird regelmäßig gepflegt, hat eine Community bestehend aus über 500 Contributors auf GitHub und passt sich durch seine deklarative Natur wunderbar an die React-Prinzipien an. Darüber hinaus ist er kompatibel sowohl mit dem Web (client- und serverseitig!) wie auch React Native. Er ist also sehr universell einsetzbar, sehr gut getestet und durch seine weite Verbreitung auch bewährt.

Dabei ist sein Interface selbst ziemlich simpel. In ca. 95% der Zeit wird man mit lediglich fünf Komponenten in Berührung kommen: `BrowserRouter`, `Link`, `Route`, `Redirect` und `Switch`. Darüber hinaus gibt es noch die imperative History API, die durch das `history`-Package, einem dünnen Layer über der nativen Browserimplementierung, wodurch diese cross-browser fähig gemacht wird, sowie die Higher Order Component `withRouter`, um für das Routing relevante Daten aus dem Router in eine Komponente hinein zu reichen.

Installiert wird der **React Router** via:

```
npm install --save react-router-dom
```

bzw.

```
yarn add react-router-dom
```

Die generelle Benutzung ist dabei wie bereits angesprochen *deklarativ*, erfolgt also in Form der oben erwähnten Komponenten. Router können dadurch innerhalb einer Anwendung an jeder beliebigen Stelle verwendet werden. Voraussetzung ist lediglich, dass der jeweilige Seitenbaum sich in einem **Router Context** befindet. Dieser existiert in einer typischen Anwendung nur ein einziges Mal und legt sich meist ganz außen um die Anwendung. Etwa in der folgenden Form:

```
import React from 'react';
import ReactDOM from 'react-dom';
import { BrowserRouter as Router } from 'react-router-dom';
```

```
const App = () => {
  return <Router>[...]</Router>;
};

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

Routen definieren

Jede Komponente innerhalb des `<Router></Router>`-Elements kann nun auf den **Router Context** zugreifen, darauf reagieren und ihn steuern. Verschiedene Routen legen wir durch die Verwendung der Route-Komponente an, die eine `path`-Prop enthalten sollte (Ausnahme: 404 Fehler-Routen) und wahlweise eine `render`-Prop oder eine `component`-Prop enthält. Der Unterschied liegt hier darin, dass der Wert der `render`-Prop eine **Funktion** sein muss, die ein valides **React-Element** zurückgibt (hier sei auch nochmal an das entsprechende Kapitel zu **Render-Props** erinnert), während die `component`-Prop eine **Komponente** (kein *Element!*) erwartet.

Also sieht eine korrekte Verwendung beider Props z.B. so aus:

```
import React from 'react';
import ReactDOM from 'react-dom';
import { BrowserRouter as Router } from 'react-router-dom';

const Example = () => <p>Example Komponente</p>;

const App = () => {
  return (
    <Router>
      <Route path="/example" component={Example} />
      <Route path="/example" render={() => <Example />} />
    </Router>
  );
};

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

In diesem Beispiel würde die `Example`-Komponente beim Aufruf der `/example` URL zweimal gerendert werden, da die Route-Komponente lediglich überprüft, ob der Pfad der aktuellen URL mit dem in der `path`-Prop angegebenen Wert übereinstimmt. Dies mag erst einmal verwunderlich klingen, lässt sich aber ganz logisch erklären.

Da React Router eben deklarativ ist, sagen wir React mit der Angabe zweier gleicher Routen eben erst einmal, dass wir auch zwei Komponenten rendern wollen, wenn die URL

State Management

Mit zunehmender Größe einer Anwendung wachsen meist auch deren Komplexität und die Daten, die mit der Anwendung verwaltet werden sollen. Also anders gesagt: Der **Application State** wird schwieriger und unübersichtlicher zu verwalten. Wann gebe ich wie welcher Komponente welche Props hinein? Wie wirken sich diese Props auf den State meiner Komponente aus und was passiert wenn ich den State in einer Komponente modifiziere?

Auch wenn React hier in den letzten Jahren zunehmend deutlich besser geworden ist und gerade mit der neuen **Context API** und dem **useReducer-Hook** einiges zur besseren Übersichtlichkeit von komplexen Datenstrukturen getan hat, ist es noch immer nicht ganz einfach, stets alle Daten und Datentransformationen im Blick zu haben. Um dieses Problem zu lösen, gibt es einige externe Tools für **globales State Management**, die sich im Ecosystem um React herum gebildet haben.

Zu den bekannteren Tools dieser Art zählt hier einerseits **MobX**, das sich selbst als *„Simple, scalable state management"* beschreibt, also als Werkzeug für *simples und skalierbares State Management*. Auf der anderen Seite haben wir **Redux**, zweifellos der „Platzhirsch" in der React-Welt, mitentwickelt unter anderem von Dan Abramov und Andrew Clark, die inzwischen auch dem offiziellen React-Team angehören. Redux bezeichnet sich als *„A predictable state container for JavaScript apps"* also als *„vorhersehbarer State Container für JavaScript-Anwendungen"*.

In diesem Kapitel soll es insbesondere um **Redux** gehen. Einerseits, weil ich selbst mit **Redux** in vielen Projekten gearbeitet habe und sehr gute Erfahrungen damit machen durfte, andererseits aber auch, weil es mit (laut npmjs.com) wöchentlich rund **4 Millionen Installationen** ganz klar deutlich mehr im Mainstream angekommen ist als **MobX**, mit immer noch respektablen, verglichen jedoch mit **Redux** doch überschaubaren **200.000 Installationen**.

Redux erfreut sich dabei noch immer an steigender Beliebtheit und hat wachsende Downloadzahlen zu verzeichnen, obwohl es bereits mehrmals von irgendwelchen Propheten totgesagt wurde. Als die finale **Context API** in **React 16.3.0** veröffentlicht wurde, hieß es, dass **Redux** damit obsolet würde (wurde es nicht), als mit **React 16.8.0** die **Hooks** und hier insbesondere der stark an **Redux** angelehnte **useReducer-Hook** eingeführt wurde, gab es diese Stimmen erneut.

In der Realität sieht das so aus, dass die Zahl der Installationen auch nach Einführung von **Context** und **Hooks** weiter steigen und **Redux** selbst intern **Gebrauch** dieser neuen Möglichkeiten macht, um einerseits die Performance zu verbessern und andererseits die Verwendung der eigenen API zu vereinfachen. Darüber hinaus hat **Redux** inzwischen ein

unheimlich großes Ecosystem an eigenen Addons und Tools um sich versammelt, das auch durch die in React neu hinzugekommenen Funktionen nicht ersetzt wird.

Einführung in Redux

Bei **Redux** handelt es sich also wie beschrieben um einen *vorhersehbaren State Container*. Doch was bedeutet das genau? An dieser Stelle möchte ich gern etwas weiter ausholen, da ich es für wichtig erachte, das Grundprinzip zu verstehen um hinterher Fehler zu vermeiden, die schnell gemacht werden, hat man das Prinzip hinter **Redux** nicht zumindest in sehr groben Ansätzen verinnerlicht.

Erst einmal haben wir mit **Redux** ein Tool, das in Teilen angelehnt ist an die Prinzipien der **Flux-Architektur**. Diese wurde - wie auch **React** selbst - von **Facebook** entwickelt, um die Entwicklung clientseitiger Web Anwendungen zu vereinfachen. Das Grundprinzip sieht dabei einen **unidirektionalen Datenfluss** vor, bei dem Daten immer nur **in eine Richtung** fließen. Also das Prinzip, wie wir es bereits aus **React** selbst kennen: eine **Aktion** (bspw. ausgelöst durch einen Button-Klick) ändert den **State**, die **State-Änderung** löst ein **Rerendering** aus und erlaubt es dann weitere Aktionen auszuführen.

Nach diesem Prinzip funktioniert auch **Redux**, mit dem entscheidenden Unterschied jedoch, dass der State statt nur innerhalb einer Komponente eben **global** verwaltet wird und somit alle Komponenten, egal wo im Seitenbaum sich diese befinden, auf **sämtliche Daten** zugreifen können.

Um Redux in einem Projekt zu nutzen, müssen wir es natürlich zuerst wieder einmal über die Kommandozeile installieren:

```
npm install redux react-redux
```

bzw. mit Yarn:

```
yarn add redux react-redux
```

Installiert werden hier **zwei** Pakete. Die Library `redux` selbst und `react-redux`. Während mit dem `redux` Paket die eigentliche State Management Library installiert wird, werden mit `react-redux` die sog. **Bindings** installiert. Auf gut Deutsch gesagt ist dies einfach nur ein Paket mit einigen **React-Komponenten** die konkret für die Verwendung von **Redux** mit **React** entwickelt und daraufhin optimiert wurden. Keine große Magie.

Theoretisch wäre auch die Verwendung von **Redux** alleine möglich, allerdings müssten wir uns dann selbst darum kümmern zu schauen, wann Komponenten neu gerendert werden und darum, wie Daten aus einer Komponente in den State Container rein und wieder raus kommen.

Da wir das nicht wollen, weil sich jemand anderes (der sich viel besser damit auskennt als wir alle) das bereits gemacht hat, nutzen wir eben zusätzlich `react-redux`.

Store, Actions und Reducer

Nicht von den möglicherweise noch unbekannten Begriffen verunsichern lassen. Wir gehen sie der Reihe nach durch und irgendwann macht es Klick.

Alle Daten in **Redux** befinden sich in einem sogenannten **Store**, der sich um die Verwaltung des globalen **States** kümmert. Theoretisch kann eine Anwendung auch mehrere Stores haben, im Konzept von Flux ist das sogar explizit so vorgesehen, in **Redux** ist das jedoch um Komplexität zu reduzieren eher unüblich und so beschränken sich React-Anwendungen, die **Redux** einsetzen, meist auch auf lediglich einen *einzigen* **Store** als **Single Source of Truth**, also als *die* einzige wahre Quelle für alle Daten. Der Store stellt Methoden bereit, um die sich in ihm befindlichen Daten zu verändern (`dispatch`), zu lesen (`getState`), und auf Änderungen zu reagieren (`subscribe`).

Die einzige Möglichkeit um Daten in einem **Store** zu verändern, ist dabei das Auslösen (*„dispatchen"*) von **Actions**. Auch hier lässt sich **Redux** wieder von Ideen aus der Flux-Architektur inspirieren und macht es erforderlich diese **Actions** im Format der **Flux Standard Actions** (FSA) zu halten. Eine solche **FSA** bestehen aus einem simplen JavaScript-Objekt, das immer **zwingend** eine `type`-Eigenschaft besitzen *muss* und darüber hinaus die drei weiteren Eigenschaften `payload`, `meta` und `error` besitzen *kann*, wobei uns in erster Linie einmal die `payload` interessieren soll, mit der wir es in 9 von 10 Fällen, in denen wir eine **Action** *dispatchen*, zu tun haben werden.

Die **Payload** stellt sozusagen den *Inhalt* einer **Action** dar und kann vom simplen Boolean oder String, über numerische Werte, bis hin zu komplexen Arrays oder Objekten beliebige Daten beinhalten, die serialisierbar sind, also in Form einer JSON-Repräsentation gespeichert werden können.

Beispiel für eine typische **Action** in **Redux**:

```
{
  type: "SET_USER",
  payload: {
    id: "d929e553-7079-4309-8c7d-2d2db39922c6",
    name: "Manuel"
  }
}
```

Wird eine **Action** durch die vom **Store** bereitgestellte `dispatch`Methode ausgelöst, wird der zum Zeitpunkt des Aufrufs aktuelle **State** zusammen mit der ausgelösten **Action** an die

Mehrsprachigkeit

Eins der Themen, das neben **Routing** und **State Management** immer wieder aufkommt, ist **Mehrsprachigkeit**. Wie bekomme ich meine Anwendung übersetzt, so dass ein deutscher Benutzer ein deutsches Interface sieht, während andere Benutzer bspw. ein englisches Interface sehen.

Ich möchte in diesem Kapitel nicht generell in die Tiefen der **Internationalisierung** (meist abgekürzt: **i18n**) einsteigen, denn hier soll es um **React** gehen. Ich setze daher ein gewisses (geringes) Grundverständnis voraus, was die Internationalisierung von User Interfaces angeht und konzentriere mich daher insbesondere darauf, wie man diese in React umsetzen kann.

Wie Mehrsprachigkeit in sehr simpler Form für einfache Apps mittels **Context API** umgesetzt werden kann, wurde bereits im Kapitel über die **Context API** demonstriert, wo wir eine sehr rudimentäre Form der Mehrsprachigkeit beispielhaft mit eben dieser API umgesetzt haben. Mit zunehmender Komplexität einer Anwendung steigt aber auch die Komplexität an die Mehrsprachigkeit. Plötzlich werden Dinge wichtig wie die Verwendung von Platzhaltern oder die Verwendung des Plurals. Hier ist es dann irgendwann sinnvoll, auf Pakete zurückzugreifen, die genau zu diesem Zweck entwickelt worden sind.

Hier gibt es einige bewährte Optionen im **React-Ecosystem,** auf die wir zurückgreifen können. Zu den bekannteren gehören hier **Lingui**, **Polyglot**, **i18next** oder **react-intl** und auch Facebook hat mit **FBT** ein eigenes Framework für die Internationalisierung von React-Anwendungen im Angebot. In diesem Kapitel geht es vorrangig um `i18next` und dessen React-Bindings `react-i18next`.

Warum? Nun, ich habe in verschiedenen Projekten mit `react-intl` und `react-i18next` gearbeitet, habe die anderen Alternativen ausführlich evaluiert und bin der festen Überzeugung, dass **i18next** in vielerlei Hinsicht die beste der genannten Alternativen darstellt. Es existiert eine sehr große und aktive Community rund um **i18next**, es werden neben **React** auch noch viele andere Frameworks, Libraries und Plattformen unterstützt, es läuft ohne großen Aufwand sowohl clientseitig im Browser als auch serverseitig in Node.js und das Übersetzungsformat von **i18next** wird von so ziemlich allen großen online Übersetzungs-Services als Exportformat angeboten.

Darüber hinaus bot es als erstes Paket bereits wenige Tage nach deren Erscheinen Unterstützung für Hooks und wurde sogar gezielt auf dessen Verwendung optimiert, ohne dabei an Abwärtskompatibilität einzubüßen. Dabei bleibt die API überwiegend simpel und einfach zu benutzen und bietet maximale Flexibilität. Um es mit anderen Worten zu sagen: Das ganze Paket ist sehr komplett und lässt wenig bis gar keine Wünsche offen.

Setup von i18next

Um es zu installieren, rufen wir wieder unser Terminal auf und geben ein:

```
npm install i18next react-i18next
```

bzw. mit Yarn:

```
yarn add i18next react-i18next
```

Wir installieren hier mit `i18next` das **Internationalisierungs-Framework** selbst, sowie mit `react-i18next` dessen **React Bindings**, also sozusagen eine Reihe an Komponenten und Funktionen, die uns die Arbeit mit i18next in React deutlich erleichtern. Nach dem Prinzip, das wir schon im Kapitel zu State Management mit `redux` und `react-redux` kennengelernt haben.

Starten wir, indem wir uns zunächst zwei Objekte anlegen, die unsere Übersetzungen beinhalten. Eins mit Texten in Deutsch, eins mit Texten in Englisch:

```
const de = {
  greeting: 'Hallo Welt!',
  headline: 'Heute lernen wir Internationalisierung mit React',
  messageCount: '{{count}} neue Nachricht',
  messageCount_plural: '{{count}} neue Nachrichten',
};

const en = {
  greeting: 'Hello world!',
  headline: 'Today we learn Internationalization with React',
  messageCount: '{{count}} new message',
  messageCount_plural: '{{count}} new messages',
};
```

Um **i18next** nun in unserer Anwendung zu verwenden, müssen wir es importieren, initialisieren und das React Plugin übergeben. Dies sollte idealerweise am Anfang unserer Anwendung passieren. Möglichst, bevor wir unsere App-Komponente an den `ReactDOM.render()`-Aufruf übergeben.

Dazu importieren wir das `i18next`-Paket selbst, sowie den benannten Export `initReactI18next` aus dem `react-i18next`-Paket:

```
import i18next from 'i18next';
import { initReactI18next } from 'react-i18next';
```

Anschließend nutzen wir die `.use()`-Methode, um das React Plugin an **i18next** zu übergeben, sowie die `.init()`-Methode, um **i18next** zu initialisieren:

```
i18next
  .use(initReactI18next)
  .init({ ... });
```

Die `init()`-Methode erwartet dabei ein Config-Objekt, das mindestens die beiden Eigenschaften `lng` (für die ausgewählte Sprache) sowie `resources` (für die Übersetzungen selbst) beinhalten sollte. Darüber hinaus ist es oft sinnvoll, eine `fallbackLng` zu erstellen, also eine Fallback-Sprache, die dann benutzt wird, wenn eine Übersetzung in der ausgewählten Sprache nicht vorhanden ist. Insgesamt bietet **i18next** hier über 30 verschiedene Konfigurationsoptionen an, doch die genannten drei sind die, die uns für den Moment interessieren:

```
i18next.use(initReactI18next).init({
  lng: 'en',
  fallbackLng: 'en',
  resources: {
    en: {
      translation: en,
    },
    de: {
      translation: de,
    },
  },
});
```

Wir setzen also die Sprache erst einmal auf Englisch, ebenso die Fallback-Sprache. Dann folgt ein `resources`-Objekt, das etwas Erläuterung bedarf. Das Objekt hat die Form:

```
[Sprache].[Namespace].[Translation Key] = Übersetzung
```

Die Sprache dürfte klar sein. Das kann `de` für Deutsch sein, `en` für Englisch oder auch `de-AT` für deutsch mit österreichischer Regionalisierung. Die Eigenschaft besitzt als Wert ein Objekt, bestehend aus mindestens einem bis zu theoretisch unbegrenzt vielen **Namespaces**.

Der **Namespace** ist ein zentrales Feature in **i18next,** das benutzt werden kann, aber nicht muss. Es erlaubt dem Entwickler, größere Übersetzungsdateien in mehrere Teile zu splitten, die bei Bedarf dynamisch nachgeladen werden können. Während dieses Feature bei kleineren Anwendungen nicht unbedingt sinnvoll ist, kann es in größeren und komplexeren Anwendungen dazu benutzt werden, Übersetzungen klein und übersichtlich zu halten, etwa indem jeder größere Seitenbereich einen eigenen **Namespace** erhält. Übersetzungsdateien für

Schlusswort

Lieber Leser, ich hoffe du hast tatsächlich etwas gelernt und verstanden, während du dieses Buch gelesen hast. Mich hat die Arbeit am Buch rund 1,5 Jahre gekostet (abgesehen von den letzten beiden Monaten vor der Veröffentlichung nicht in Vollzeit natürlich). In dieser Zeit ist viel passiert, viele Dinge haben sich geändert und zwar so signifikant, dass ich den Release des Buchs mehrmals um einige Tage bis Wochen verschoben habe, weil ich nicht gleich am Tag der Veröffentlichung veraltetes Material im Buch haben wollte.

Leider sind dadurch auch einige Themen zu kurz gekommen oder noch gar nicht beschrieben worden, denen ich eigentlich mehr Platz in diesem Buch einräumen wollte, die aber andererseits auch nicht unbedingt ein so zentraler Bestandteil von React sind, als dass ich deswegen den Release noch weiter hinauszögern wollte. Die Optimierung des Build-Setups mit Webpack ist eins dieser Themen. Zu dem Thema habe ich ein anderes Buch auf Amazon gefunden mit knapp 500 Seiten. Das hätte hier definitiv den Rahmen gesprengt.

Andere Themen wie Server Side Rendering habe ich ebenfalls nur am Rande erwähnt. Zwar finde ich das Thema wichtig, hier habe ich nach einigen Projekten allerdings den Eindruck gewonnen, dass der Nutzen von Server Side Rendering nicht immer unbedingt die Kosten aufwiegt.

Dieses Buch wird mit der Veröffentlichung auch öffentlich auf GitHub verfügbar sein. Hier werde ich in den Tagen nach dem Release einige Issues erstellen, mit denen ihr gern darüber abstimmen könnt, welche Themen, die im Buch bisher noch fehlen, wichtig sind. Ich werde das Buch dann nach Bedarf nach und nach um neue Kapitel und weitere Inhalte erweitern.

Übrigens freue ich mich auch über Issues[44], wann immer ihr einen Fehler im Buch entdeckt. Egal ob Rechtschreibfehler, Grammatik oder inhatlicher Natur. Da ich das Buch im Selbstverlag veröffentliche, hatte ich keinen Zugriff auf umfassendes Lektorat, wie es von den großen Verlagen gestellt wird. Jedoch wollte ich die volle Kontrolle über sämtliche Rechte und Veröffentlichungskanäle behalten. Und dazu gehört auch die kostenlose Online-Version unter https://lernen.react-js.dev/ – dies wäre mit vielen Verlagen leider nicht möglich gewesen.

Ich habe durch React in den vergangenen Jahren viele spannende Projekte begleitet, habe immens viel über Frontend- und UI-Entwicklung gelernt und nicht zuletzt auch viele großartige Leute durch die Arbeit mit React kennengelernt. Diese Erfahrungen wollte ich einfach ein Stück weit an die Community zurückgeben, weshalb die HTML-Version kostenlos ist und auch immer bleiben wird!

Wer mich und meine Arbeit unterstützen will, kann sich für einen – wie ich finde – sehr fairen Preis auch das eBook für Kindle, Apple Books oder Google Play Books kaufen und das Buch bequem auf seinem bevorzugten Lese-Gerät lesen. Und natürlich freue ich mich auch immer über interessante Anfragen zu Projekten mit dem Schwerpunkt React (klar, was sonst?) in Berlin.

Mehr über mich gibt es auf meiner Website: https://www.manuelbieh.de/ – die, wie sollte es anders sein, mit Gatsby entwickelt wurde.

Ich freue mich über jedes Feedback von euch, egal ob positiv oder negativ! Dazu erreicht ihr mich auch auf Twitter unter @manuelbieh.

Links

- [1] https://github.com/manuelbieh/react-lernen/issues
- [2] https://github.com/manuelbieh/react-lernen
- [3] https://www.gitbook.com/
- [4] https://github.com/reactjs/rfcs
- [5] https://github.com/reactjs/react-codemod
- [6] https://github.com/facebook/react/issues?utf8=%E2%9C%93&q=is%3Aissue%20is%3Aopen%20umbrella
- [7] https://github.com/creationix/nvm
- [8] https://github.com/coreybutler/nvm-windows
- [9] https://atom.io/packages/language-babel
- [10] https://marketplace.visualstudio.com/items?itemName=dzannotti.vscode-babel-coloring
- [11] https://github.com/babel/babel-sublime
- [12] https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi
- [13] https://addons.mozilla.org/de/firefox/addon/react-devtools/
- [14] https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd
- [15] https://addons.mozilla.org/de/firefox/addon/remotedev/?
- [16] https://www.javascriptstuff.com/react-starter-projects/
- [17] https://docs.npmjs.com/files/package.json#name
- [18] https://codesandbox.io/
- [19] https://codesandbox.io/s/new
- [20] https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Promise
- [21] https://developer.mozilla.org/en-US/docs/Web/API/HTMLLabelElement/htmlFor
- [22] https://www.w3.org/TR/DOM-Level-3-Events/
- [23] https://michelebertoli.github.io/css-in-js/
- [24] https://www.styled-components.com/docs/basics
- [25] https://github.com/emotion-js/emotion
- [26] https://github.com/zeit/styled-jsx
- [27] https://github.com/cssinjs/jss/tree/master/packages/react-jss
- [28] https://github.com/FormidableLabs/radium
- [29] https://github.com/callstack/linaria
- [30] https://github.com/reduxjs/reselect
- [31] https://www.i18next.com/overview/configuration-options

- [32] https://locize.com/
- [33] https://www.lokalise.co
- [34] https://react.statuscode.com/
- [35] https://this-week-in-react.org/
- [36] http://reactjsnewsletter.com/
- [37] http://newsletter.fullstackreact.com
- [38] https://reactdigest.net/
- [39] https://tinyreact.email/
- [40] https://undefined.fm/
- [41] https://reactpodcast.com/
- [42] https://github.com/chantastic
- [43] https://syntax.fm/
- [44] https://github.com/manuelbieh/react-book/issues