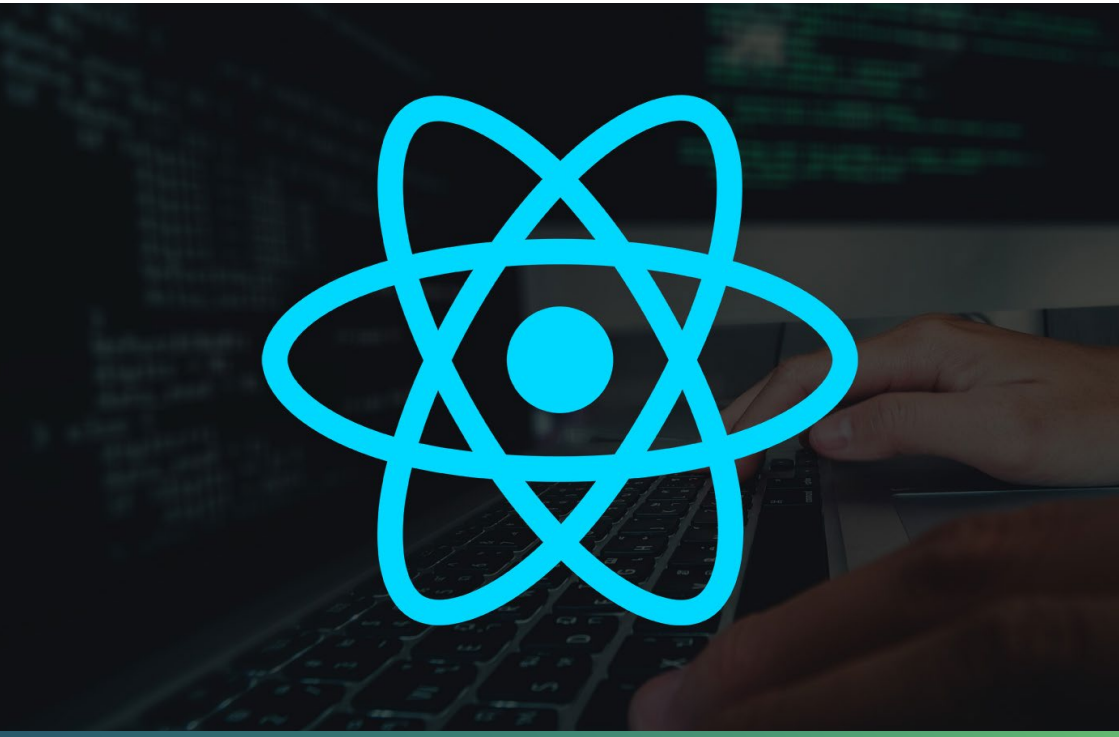
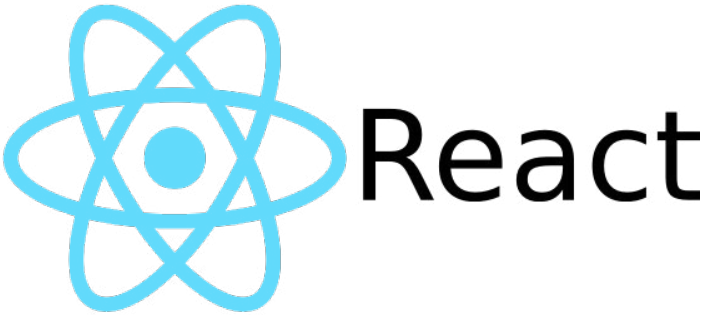
Host Europe



React

Experten-Tipps und neue Techniken für Ihre Projekte

E-Book


React

Inhaltsverzeichnis

Der Redakteur

Oliver Lindberg

Oliver Lindberg ist ein unabhängiger Redakteur, Content-Consultant, und Gründer von Pixel Pioneers, einer Konferenz für Frontend-Entwickler und UX/UI Designer. Ehemals Chefredakteur der wegweisenden Zeitschrift 'net magazine', beschäftigt Oliver sich inzwischen seit mehr als 15 Jahren mit Webdesign und -entwicklung und hilft internationalen Unternehmen bei der Umsetzung von erfolgreichen Content(-Marketing)-Strategien.

Vorwort

React – 2013 von Facebook ins Leben gerufen – ist eine der am häufigsten verwendeten JavaScript-Bibliotheken und hat das Erstellen von Benutzeroberflächen und UI-Komponenten wesentlich beeinflusst. Die Vorteile liegen in der einfachen Handhabung, der daraus folgenden gesteigerten Produktivität, besseren Code-Qualität, SEO-Freundlichkeit und in der virtuellen DOM, die es erlaubt, schnelle und skalierbare Applikationen zu bauen. Heute gibt es um die 1.5 Millionen Webseiten, die mit React entwickelt worden sind, darunter Netflix, Uber, Airbnb und die New York Times.

Grund genug also, sich die berühmte Open Source-Bibliothek einmal genauer anzusehen. In diesem E-Book führen wir Ihnen neue Techniken vor, die Ihnen bei der Anwendung auf ihre eigenen Projekte die Arbeit erleichtern. Als Autoren haben wir – wie auch schon für unser Angular-E-Book – wieder die besten Experten gewonnen, die es im deutschsprachigen Raum gibt.

Zu Beginn präsentiert Sebastian Springer einen Überblick auf Reacts modulare Komponentenarchitektur und die sogenannten React Patterns. Dabei spricht er Themen an, die in den folgenden Kapiteln aufgegriffen und tiefgehender behandelt werden. So beschäftigt sich Manuel Mauky mit React Hooks, Nils Hartmann zeigt uns, wie man React-Anwendungen typsicher mit TypeScript bauen kann, um potentielle Programmierfehler bereits während der Entwicklung zu erkennen und zu vermeiden, und Lisa Oppermann widmet sich dem globalen State-Management von Redux, mit dem man Applikationen übersichtlicher gestalten kann. Abschließend erläutert Jan Bussieck die Optimierungen für den Bau komplexer UIs, die uns der neue Kernalgorithmus React Fiber ermöglicht.

Viel Spaß beim Lesen und besonders bei der Anwendung in der Praxis!

Der Autor

Sebastian Springer

Sebastian Springer ist JavaScript-Entwickler bei Maiborn Wolff in München und beschäftigt sich vor allem mit der Architektur von client- und server-seitigem JavaScript. Er ist Berater und Dozent für JavaScript und vermittelt sein Wissen regelmäßig auf nationalen und internationalen Konferenzen.

React Patterns

Im Vergleich zu anderen Frontend-Frameworks wie Angular oder Vue bietet React relativ viele Freiheiten und macht wenig Vorgaben hinsichtlich des Aufbaus und der Architektur einer Applikation. Für fortgeschrittene Entwickler stellt das weniger ein Problem dar. Für Einsteiger bedeutet es jedoch eine vergleichsweise hohe Einstiegshürde. Aus diesem Grund haben sich verschiedene Muster etabliert, die die Strukturierung einer Applikation erleichtern.

Dieses Kapitel befasst sich mit einigen der wichtigsten Entwurfsmustern aus dem React-Universum. Den Anfang macht das fundamentalste Merkmal einer React-Applikation: die Komponenten-Architektur.

Der komponentenorientierte Ansatz von React

Allen React-Applikationen liegt eine komponentenorientierte Architektur zugrunde. Diese beeinflusst den grundlegenden Aufbau einer Applikation entscheidend. Eine Applikation, die diesem Architekturansatz folgt, ist wie ein Baum aufgebaut. Eine Wurzelkomponente bildet die Basis und verzweigt sich in eine Hierarchie von Kind-Komponenten.

Bei der Implementierung besteht der erste Arbeitsschritt daraus, die gewünschte Oberfläche in eine Komponentenhierarchie zu zerlegen. Eine solche Komponentenhierarchie ist in den seltensten Fällen frei von Zuständen. Eine Tabellenzeile kann beispielsweise hervorgehoben, ein Datensatz inaktiv oder ein Button versteckt sein. Diese Zustände werden in Form eines Komponenten-States innerhalb der Hierarchie gespeichert. Die Platzierung des States ist abhängig davon, welche Komponenten die Informationen benötigen. Im einfachsten Fall könnte der gesamte State

der Applikation in der Wurzelkomponente abgelegt werden. Dies hätte jedoch den Nachteil, dass die Informationen an die Kind-Komponente weitergereicht werden müssen, die sie zur Anzeige benötigt.

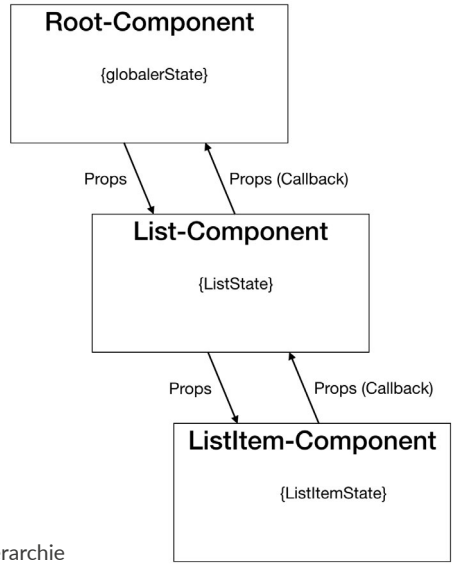
Bei einer Tiefe von zwei oder drei Komponenten-Ebenen ist das Durchreichen von State-Informationen noch praktikabel. Wird der Baum jedoch tiefer, stößt man schnell an die Grenzen der Übersichtlichkeit und Wartbarkeit. Aus diesem Grund sollten Sie darauf achten, den State nicht unnötig weit von der Stelle zu platzieren, an der er benötigt wird. Die Verteilung der State-Informationen erfolgt über die sogenannten Props. Das sind im weitesten Sinne so etwas wie die Parameter einer Komponente. Sie können das Verhalten einer Komponente von außen beeinflussen und tragen damit zu einer besseren Wiederverwendbarkeit einer Komponente bei. Props und State sind zwei fundamentale Elemente in einer komponentenbasierten Architektur. Aus diesem Grund finden Sie diese beiden Konstrukte nicht nur in React, sondern auch in den anderen wichtigen Frontend-Frameworks.

Sowohl Angular als auch Vue verfolgen ebenfalls einen komponentenbasierten Ansatz beim Aufbau eines Web-Frontends. Jedoch geht React hier einen bemerkenswerten Weg: Das Erzeugen einer Komponente ist sehr einfach und leichtgewichtig. Die einzigen Anforderungen an eine solche React-Komponente sind, dass es sich um eine Funktion handeln muss und dass diese eine JSX-Struktur zurückgeben muss. Zwar ist es auch möglich, Komponenten auf Basis von JavaScript-Klassen zu formulieren, in modernen React-Applikationen ist diese Art der Komponenten jedoch kaum noch anzutreffen.

Da es React Ihnen als Entwickler so einfach macht, neue Komponenten zu erzeugen, sind React-Komponenten tendenziell kleiner als in den anderen Frameworks. Sie sollten Ihre Komponenten nach Möglichkeit auch mit einem Maximum an Wiederverwendbarkeit designen. Damit reduzieren

Sie zum einen die Anzahl der Komponenten in einer Applikation, was eine bessere Übersicht schafft, zum anderen können Sie solche wiederverwendbaren Komponenten auch in anderen Applikationen nutzen. Mit der Zeit schaffen Sie sich so eine Bibliothek aus häufig verwendeten Komponenten, die Ihnen gerade bei Standardaufgaben sehr viel Arbeit abnehmen.

Die Grundlage einer wiederverwendbaren Komponente ist das bereits erwähnte Konzept der Props. Mit diesen Parametern der Komponente können Sie beispielsweise einen Datensatz an die Komponente übergeben, den Sie rendern möchten, oder Konfigurationsinformationen, die das Aussehen der Komponente bestimmen. Die Props einer Komponente können von beliebigen Datentypen sein. Es ist sowohl erlaubt, einfache Datentypen als auch Objekte und Arrays an eine Komponente zu übergeben. Auch Funktionen kommen zum Einsatz und zwar vor allem, wenn eine Komponente mit Ihrer Elternkomponente kommunizieren muss.



Komponentenhierarchie

Die Props einer Komponente haben noch einen weiteren Zweck, als den reinen Datenfluss in einer Komponentenhierarchie abzubilden. Eine Änderung der Props führt automatisch dazu, dass die Komponente neu gerendert wird.

TypeScript

Mit den Props wird auch gleich eine potenzielle Schwäche der Kombination aus nativem JavaScript und React deutlich: Der Entwickler einer Komponente kann keine Typen für die Props festlegen. Je mehr Personen an einer Applikation arbeiten, desto deutlicher wird das Problem, da kein direkter Austausch zwischen den einzelnen Entwicklern stattfindet und es tendenziell auch zu viele Komponenten gibt, um den Überblick zu behalten. Dieses Problem kann einfach mit der prop-types-Bibliothek gelöst werden.

Eine bessere, weil umfassendere Lösung, bietet der Einsatz eines Typsystems, und hier ist die am häufigsten anzutreffende Variante TypeScript. Mittlerweile unterstützt Create React App die Initialisierung einer React-Applikation mit TypeScript, so dass bereits zu Beginn mit TypeScript gearbeitet werden kann und auch schon eine passende Konfiguration aller Entwicklungs- und Build-Werkzeuge vorliegt. Nachfolgend sehen Sie ein Beispiel einer einfachen Komponente, die TypeScript-Features wie beispielsweise die Typisierung der Props nutzt.

```tsx
import React from 'react';
import Address from './address';

interface Props {
  address: Address;
  onDelete: (id: number) => void;
}

const ListItem: React.FC<Props> = ({ address, onDelete }) => {
  <div>
    <div>{address.name}</div>
    <div>{address.street}</div>
    <div>{address.place}</div>
    <div>{address.country}</div>
    <div>
      <button onClick={() => onDelete(address.id)}>delete</button>
    </div>
  </div>;
};

export default ListItem;
```

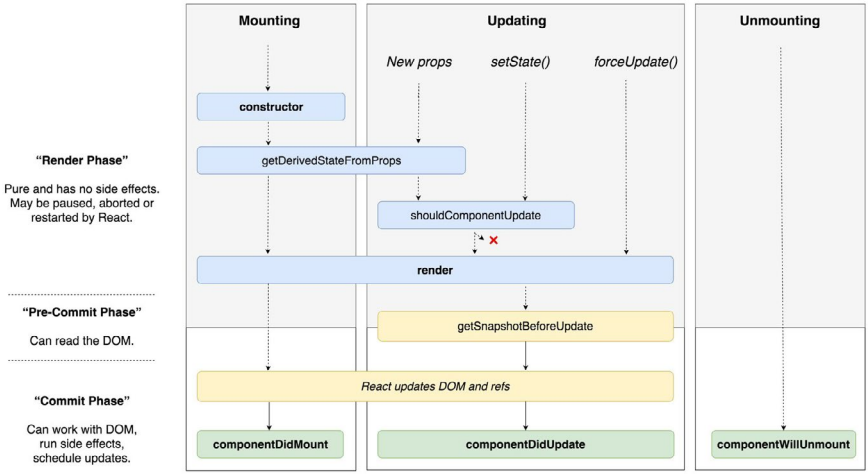
Zum Code →

Komposition

React hat seit dem Release der Version 16 eine rasante Entwicklung durchgemacht. Nachdem das Entwicklerteam den Fiber Reconciler vorgestellt hat, wurden im Zuge einer Reihe von Minor-Releases zahlreiche neue Features eingeführt, die die Entwicklung von Applikationen entscheidend beeinflussen.

Das wohl wichtigste Feature war die Einführung der Hook-API. Mit ihr hat das Entwicklerteam von React die Funktionskomponenten auf Augenhöhe mit den Klassenkomponenten gebracht. Dabei gibt es einige Vorteile, die für den Einsatz von Funktionskomponenten sprechen: Sie kommen ohne das ***this***-Schlüsselwort aus, außerdem können State und Lifecycle-Methoden aufgeteilt werden. Innerhalb einer Klassenkomponente können Sie über ***this*** auf die aktuelle Instanz der Klasse zugreifen. Innerhalb von Callback-Funktionen, wie sie beispielsweise bei Eventhandlern zum Einsatz kommen, verweist ***this*** jedoch nicht mehr auf das Objekt.

Dieser Umstand ist eine häufige Fehlerquelle und kann beispielsweise verhindert werden, wenn Sie die Methoden in den Objektkontext binden oder statt regulärer Methoden Eigenschaften mit Arrow-Funktionen als Wert verwenden. Ein weiterer Nachteil bei den Klassenkomponenten ist, dass es keine Möglichkeit gibt, den State und die Lifecycle-Methoden einer Klassenkomponente zu unterteilen. Mit der Kombination aus Funktionskomponenten und der Hook-API wird es möglich, mehrere State-Objekte innerhalb einer Komponente vorzuhalten. Damit können Sie beispielsweise die Zustände von mehreren GUI-Elementen sauber voneinander trennen. Ähnliches gilt für die Lifecycle. Wo Sie bei einer Klasse nur beispielsweise einmal die ***componentDidMount***-Methode implementieren können, wird es dank des Effect-Hooks möglich, mehrere Funktionen für den gleichen Lifecycle-Abschnitt zu implementieren.



Der Lifecycle einer Komponente (erstellt von Dan Abramov)

Die Einführung der Hook-API hat insgesamt dazu geführt, dass die Strukturen einer React-Applikation besser aufgeteilt werden können und sich bestimmte Codeblöcke nur noch um eine Sache kümmern. Die Custom Hooks, eine Kombination aus den Builtin-Hooks von React und eigener Implementierung, erlauben es, die Applikationslogik von der Komponente zu trennen. Dies dient zum einen dazu, den Code einer Komponente kleiner zu halten, und zum anderen – und das ist der eigentliche Verwendungszweck von Custom Hooks – können Sie damit wiederverwendbare Hooks erzeugen. So können Sie beispielsweise einen konfigurierbaren Custom Hook implementieren, der beim Laden der Komponente Daten vom Server lädt und im lokalen State festhält.

Hier sehen Sie ein Beispiel für einen solchen Custom Hook:

```tsx
import { useState, useEffect } from 'react';

export default function useData<T>(url: string): T[] {
  const [state, setState] = useState<T[]>([]);

  useEffect(() => {
    async function getData() {
      const response = await fetch(url);
      const data = await response.json();
      setState(data);
    }
    getData();
  }, []);

  return state;
}
```

Zum Code →

Der allgemeinen Konvention folgend beginnt der Name der Hook-Funktion mit dem Präfix *use*. Wann immer Sie also in React eine Funktion, die mit *use* beginnt, nutzen, ist dies ein klarer Hinweise darauf, dass es sich dabei um eine Hook-Funktion handelt. Wie jeden Custom Hook können Sie auch die Funktion aus dem Beispiel in einer beliebigen Funktionskomponente einbinden und müssen beim Aufruf lediglich die URL des serverseitigen Endpunkts übergeben, von dem die Daten bezogen werden sollen. Da der Custom Hook als TypeScript Generic implementiert ist, haben Sie auch die vollständige Unterstützung des TypeScript-Compilers und Ihrer Entwicklungsumgebung.

Dieser Hook lässt sich noch erweitern, so dass er beispielsweise Funktionen zum Modifizieren oder Löschen einzelner Datensätze zur Verfügung stellt. Auch dies sind Standardaufgaben, die sich bis auf die Datenstrukturen, auf denen sie arbeiten, nicht sonderlich unterscheiden.

Die meisten Bibliotheken im React-Ökosystem haben in der Zwischenzeit auf Hooks umgestellt oder bieten zumindest eine Hook-Implementierung an. Ein recht prominentes Beispiel ist hier Redux. Die zentrale State-Management-Bibliothek fühlt sich dank der Hook-API nicht mehr wie ein Fremdkörper an, sondern gliedert sich nahtlos in das Frontend ein.

Patterns aus der alten Welt neu interpretiert

Zwei Patterns, die React über eine lange Zeit begleitet haben, sind Render Props und Higher Order Components, oder kurz HoC. Beide Muster dienen dazu, eine Komponente mit zusätzlichen Informationen anzureichern. In der Implementierung von einfachen Applikationen kommen sowohl Render Props als auch HoCs eher selten vor. Dafür waren sie vor dem Aufkommen der Hook API das Quasi-Standardmittel von Bibliotheken, um ihre Features zur Verfügung zu stellen. Bei den Render Props wird eine Komponente implementiert, die entweder eine ***render***-Prop aufweist, daher der Name, oder die über die ***children***-Prop auf ihr Kindelement zugreift, das in diesem Fall eine Funktion ist. Die außenliegende Komponente implementiert wiederverwendbare Logik und übergibt die Informationen beziehungsweise Funktionen an die ***render***-Funktion.

Bei einer HoC handelt es sich um eine Funktion, die eine Komponente als Argument erhält und eine Komponente zurückgibt. Dieses Muster verfolgt den gleichen Zweck wie schon die Render Props mit dem Unterschied, dass hier etwas deutlicher wird, dass durch die HoC eine

Eingabekomponente angereichert wird. Typischerweise beginnen HoC-Funktionen mit dem Präfix ***with***. Ein prominentes Beispiel einer solchen Implementierung ist der React Router. Die ***withRouter***-Funktion ist genau eine solche HoC-Funktion. Übergeben Sie eine Ihrer Komponenten an diese Funktion, wird sie um Routing-Informationen angereichert, und Sie können über die Props Ihrer Komponente beispielsweise auf den aktuellen Pfad und Routing-Parameter zugreifen.

Der Vorteil von Render Props und HoC ist, dass beide im Gegensatz zu Hooks mit Klassenkomponenten funktionieren und auch mit Funktionskomponenten verwendet werden können. Beide Muster werden jedoch zunehmend von Hook-Implementierungen verdrängt. Am Beispiel des React Routers und der ***useParams***-Funktion sehen Sie, dass Sie statt über die HoC-Funktion über eine Hook-Funktion auf die Routing-Parameter zugreifen können.

Anordnung und Aufbau einer Applikation

Wie schon erwähnt, macht React kaum Vorgaben. Das gilt sowohl für die Form des Quellcodes als auch die Platzierung des Quellcodes im Dateisystem. Theoretisch können Sie sämtliche Komponenten, Hooks und alle weiteren Strukturen Ihrer Applikation in eine einzige Datei ablegen. Beim Build der Applikation optimieren Werkzeuge wie Webpack den Quellcode und fassen die Komponenten in eine, beziehungsweise wenige Dateien.

Grundsätzlich sollten Sie darauf achten, dass der Quellcode Ihrer Applikation auf Lesbarkeit hin optimiert ist. Die Performance-Vorteile, die Sie durch eine übermäßige Optimierung herausholen, gehen sehr auf Kosten der Lesbarkeit. Das bedeutet, dass es teurer wird, Ihre Applikation zu erweitern oder Fehler zu beheben. Deshalb hat sich „eine Komponente pro Datei“ als Best Practice etabliert.

Verfügt Ihre Komponente über Props, können Sie das Props-Interface innerhalb der Komponenten-Datei definieren, da das Interface in der Regel an keiner weiteren Stelle benötigt wird. Aus diesem Grund wird dieses Interface auch nicht exportiert. Beim Export der Komponente können Sie zwischen default und named Export wählen. Wichtig ist an dieser Stelle nur, dass Sie innerhalb Ihrer Applikation konsistent bleiben und überall die gleiche Art verwenden.

Das Styling Ihrer Komponenten können Sie ebenfalls in der Komponenten-Datei selbst vornehmen. Je nachdem, wie viel komponentenspezifisches Styling in Ihrer Applikation notwendig ist, kann durchaus eine größere Menge an Styling-Code entstehen. Dies macht die Komponenten-Datei unübersichtlich, was für die Auslagerung des Stylings in eine separate Datei spricht.

Wählen Sie diesen Weg, sollten Sie darauf achten, dass die Styles möglichst nah bei der Komponenten-Datei abgelegt werden. Für die Unit-Tests Ihrer Komponenten gilt übrigens das gleiche: Legen Sie die Testdatei bei Ihrer Komponente ab. Damit haben Sie alle Dateien, die die Komponente betreffen, in einem Verzeichnis und müssen nicht suchen, beziehungsweise das Verzeichnis wechseln.

Für die Benennung der Dateien sollten Sie ebenfalls ein einheitliches Schema wählen. Häufig kommt entweder CamelCase oder Kebab-Case als Schreibweise zum Einsatz, wobei Zweiteres Probleme mit case-sensitiven Dateisystemen vermeidet. Die Komponentendateien enden mit .tsx. Die Styling-Dateien heißen in der Regel wie die Komponenten-Dateien und unterscheiden sich lediglich durch die Dateiendung. Verwenden Sie beispielsweise SCSS, lautet die Endung .scss. Die Testdatei heißt ebenfalls wie die Komponentendatei und endet auf .spec.ts.

Für die Aufteilung der Dateien in Verzeichnisse gilt, dass Sie bei umfangreicheren Applikationen mit einer fachlichen Trennung starten sollten. Haben Sie beispielsweise einen Bereich in Ihrer Applikation, der sich mit Benutzermanagement beschäftigt, erzeugen Sie ein Verzeichnis „user“. Hier finden sich alle Dateien, die mit diesem Bereich zu tun haben. Als Daumenregel gilt, dass Sie ein Verzeichnis in Unterverzeichnisse unterteilen sollten, sobald in dem Verzeichnis mehr als sieben bis zehn Dateien liegen, da ab dieser Zahl die Übersicht beginnt verloren zu gehen.

Fazit

React ist eine Frontend-Bibliothek, die Ihnen als Entwickler sehr viele Freiheiten bietet. Aus diesem Grund ist es wichtig, bei der Implementierung einer Applikation noch mehr auf die Einhaltung von Konventionen zu achten. Ein wichtiger Baustein sind dabei die Strukturen, mit denen bestimmte Arten von Problemen gelöst werden. Der konsistente Einsatz solcher Lösungsmuster verbessert die Wartbarkeit einer Applikation, da sich die Entwickler einfacher und schneller zurechtfinden. Mit der Einführung der Hook API in React haben sich viele dieser Muster geändert. Generell geht der Trend in Richtung Komposition von Applikationslogik und kleinerer wiederverwendbarer Einheiten.

Der Autor

Manuel Mauky

Manuel Mauky ist Softwareentwickler bei der Zeiss Digital Innovation in Görlitz. Er beschäftigt sich dort vor allem mit Java- und Webentwicklung sowie Softwarearchitektur. Daneben interessiert er sich für funktionale Programmierung. Er spricht regelmäßig auf Konferenzen und User-Groups und ist Organisator der Java-User-Group Görlitz.

React Hooks

Mit Version 16.8 von React hat ein Feature Einzug gehalten, welches die Art und Weise, wie man mit React Komponenten erstellt, fundamental verändert hat: React Hooks. In diesem Artikel wollen wir uns anschauen, was Hooks sind und was sie so spannend macht.

In React gibt es im Prinzip zwei Wege, um Komponenten zu bauen: Klassenkomponenten und Funktionskomponenten. Klassenkomponenten leiten sich von ***React.Component*** ab und überschreiben die ***render***-Methode. Diese Methode liefert eine DOM-Beschreibung der Komponente zurück und wird immer dann aufgerufen, wenn sich am Zustand der Komponente etwas geändert hat. Von außen können beliebige Daten in die Komponente hereingereicht werden, die innerhalb der Komponente mittels ***this.props*** erreichbar sind.

Die zweite Variante sind Funktionskomponenten, die lediglich aus einer Funktion bestehen, die ähnlich wie die ***render***-Methode funktioniert. Der Unterschied ist, dass hier die „Props“ direkt als Parameter der Funktion übergeben werden. Von außen lässt sich bei der Benutzung einer React-Komponente nicht erkennen, ob diese als Klassen- oder Funktionskomponente implementiert wurde.

React selbst ist geprägt von Konzepten der „Funktionalen Programmierung“ und auch der Community merkt man eine Affinität zu diesem Paradigma an. Daher verwundert es nicht, dass sich besonders die Funktionskomponenten einer besonderen Beliebtheit erfreuen. Allerdings waren Funktionskomponenten bisher in ihren Möglichkeiten beschränkt, und für viele Dinge waren stets Klassenkomponenten notwendig.

Zum einen ermöglichten nur Klassenkomponenten die Verwaltung eines lokalen Zustands, während Funktionskomponenten nur die von außen hereingereichten Daten zur Anzeige bringen konnten. Außerdem haben nur Klassenkomponenten Zugriff auf die zahlreichen Lifecycle-Methoden, um beispielsweise Code auszuführen, sobald die Komponente in den Komponenten-Baum eingehangen wurde. Diese Lifecycles sind aber notwendig, um Seiteneffekte wie Netzwerk-Requests oder Timer sauber benutzen zu können.

Dies führte in der React-Community zu einer häufig anzutreffenden Unterscheidung zwischen „dummen“ Anzeige-Komponenten, die lediglich Daten entgegennehmen und hübsch darstellen, und „smarten“ Komponenten, die selber wenig zur Darstellung beitragen, aber die Daten verwalten und Logik implementieren. Auch mit Hooks gibt es gute Gründe, diese konzeptionelle Trennung beizubehalten, jedoch war bisher eben auch die Art der Implementierung notwendigerweise unterschiedlich.

Mit Hooks gibt es nun die Möglichkeit, fast alle Aspekte, für die bisher Klassenkomponenten notwendig waren, auch mit Funktionskomponenten umzusetzen. Am besten lässt sich dies am Beispiel des lokalen Zustands verdeutlichen. In Abbildung 1 ist eine Komponente zu sehen, die einen Titel und einen Text anzeigt, wobei der Text mittels eines Buttons ein- und ausgeklappt werden kann. Der dazugehörige Code, implementiert als Klassenkomponente, findet sich in Code-Block 1. Als Sprache wurde (wie auch für alle folgenden Beispiele) TypeScript gewählt.

Abbildung 1

```tsx
import React from "react"
import "./expansion-box.css"

type Props = {
  title: string
}

type State = {
  expanded: boolean
}

export class ExpansionBoxClass extends React.Component<Props, State> {
  constructor(props: Props) {
    super(props)

    this.state = {
      expanded: false,
    }
  }

  private switchExpansion() {
    this.setState((prevState) => ({ expanded: !prevState.expanded }))
  }

  render() {
    return (
      <div className={`expansion-box ${this.state.expanded ? "expanded" : ""}`}>
        <header>
          <h1>{this.props.title}</h1>
          <button onClick={() => this.switchExpansion()}>
            {this.state.expanded ? <span>&#x25B2;</span> : <span>&#x25BC;</span>}
          </button>
        </header>

        {this.state.expanded && <section>{this.props.children}</section>}
      </div>
    )
  }
}
```

Zum Code →

CODE-BLOCK 1

Interessant ist vor allem die ***switchExpansion***-Methode. Hier wird die React-Methode ***setState*** aufgerufen, um den lokalen Zustand des ***expanded***-Flag umzuschalten. In Abhängigkeit dieses Flags wird in der ***render***-Methode die Content-Section gerendert sowie eine CSS-Klasse gesetzt.

State Hook

Die Variante als Funktionskomponente mit Hooks ist im Code-Block 2 zu sehen. Spannend ist hier zunächst, dass die Zeile X in der ***useState*** aufgerufen wird. Dies ist der erste React-Hook. Hooks sind spezielle Funktionen, deren Parameter und Rückgabe-Werte variieren können. Im Fall von useState liefert die Funktion ein Array mit zwei Elementen zurück: Das erste ist der aktuelle Wert der Zustandsvariable. Dieser kann im Return-Block genutzt werden, um die Darstellung zu beeinflussen. Genau wie bei Klassenkomponenten darf der Zustand nicht direkt mittels Zuweisung verändert werden, sondern es muss eine spezielle Funktion aufgerufen werden (bei Klassenkomponenten ***setState***), damit React von dieser Änderung Notiz nehmen und ein Neuzeichnen der Komponente veranlassen kann. Der ***useState***-Hook liefert eine solche Update-Funktion als zweites Array-Element zurück. Nicht verwirren lassen sollte man sich von den eckigen Klammern auf der linken Seite des Gleichheitszeichens. Dies ist das so genannte „destructuring“, welches TypeScript und modernes JavaScript mitbringt, um direkt auf Array-Elemente zuzugreifen und diese lokalen Variablen zuzuweisen.

```tsx
import React, { FC, useState } from "react"
import "./expansion-box.css"

type Props = {
  title: string
}

export const ExpansionBoxFunction: FC<Props> = (props) => {
  const [expanded, setExpanded] = useState(false)

  const switchExpansion = () => {
    setExpanded(!expanded)
  }

  return (
    <div className={`expansion-box ${expanded ? "expanded" : ""}`}>
      <header>
        <h1>{props.title}</h1>
        <button onClick={switchExpansion}>{expanded ? <span>&#x25B2;</span> : <span>&#x25BC;</span>}</button>
      </header>

      {expanded && <section>{props.children}</section>}
    </div>
  )
}
```

Zum Code →

CODE-BLOCK 2

Bei Funktionskomponenten gibt es keine separate ***render***-Methode. Stattdessen wird die gesamte Funktion bei Bedarf neu ausgeführt. Dies ist beispielsweise der Fall, wenn sich die von außen hinein gereichten Props ändern, oder aber, wenn die von ***useState*** als zurückgegebene Update-Funktion aufgerufen wurde. In diesem Fall liefert ***useState*** den aktualisierten Wert als erstes Array-Element, so dass innerhalb der Komponente entsprechend darauf reagiert werden kann.

Alle mit React von Hause aus mitgelieferte Hooks folgen der Namenskonvention „useX“, und diese Konvention wird auch für Custom-Hooks dringend empfohlen, um Hooks auf den ersten Blick als solche erkennen zu können. Dies ist wichtig, da für Hooks besondere Regeln existieren, die unbedingt eingehalten werden müssen:

Hooks dürfen nur in Funktionskomponenten oder Custom-Hooks, nicht jedoch in Klassenkomponenten oder sonstigen Funktionen genutzt werden. Außerdem dürfen Hooks nicht innerhalb von IF-Verzweigungen, Loops oder Unter-Funktionen aufgerufen werden, sondern müssen auf der obersten Ebene der Funktionskomponente bzw. des Custom-Hooks stehen. Der Grund für diese Regel ist, dass React die Reihenfolge der Hook-Aufrufe verfolgt und nur so Hooks über mehrere Rendering-Zyklen zuordnen kann. In der Praxis erweisen sich diese Regeln aber als unproblematisch und man gewöhnt sich recht schnell daran. Als gute Praxis hat es sich darüber hinaus erwiesen, Hooks stets am Anfang der Funktionskomponente aufzurufen, um einen guten Überblick zu behalten.

Effect Hook

Der zweite wichtige Hook, den React mitbringt, ist ***useEffect***. Er kommt vor allem in solchen Situationen zum Einsatz, die bei Klassenkomponenten mithilfe der Lifecycle-Methoden implementiert wurden. Ein Beispiel ist das asynchrone Laden von Daten über das Netzwerk, sobald die Komponente angezeigt wird: Bei Klassenkomponenten wird dazu die Methode ***componentDidMount*** implementiert, welche, wie der Name schon sagt, aufgerufen wird, direkt nachdem die Komponente in den Komponenten-Baum eingefügt wurde. In dieser Methode kann der Netzwerk-Request gestartet und auf das Ergebnis subscribed werden. Sobald das Ergebnis da ist, wird mittels ***setState*** der lokale Zustand aktualisiert, was wiederum ein Neuzeichnen der Komponente mit den geladenen Daten auslöst.

```tsx
import React from "react"

export class LoadDataClass extends React.Component {
  componentDidMount(): void {
    fetch("/some-data.json")
      .then((response) => response.json())
      .then((data) => {
        this.setState({ data })
      })
  }

  render() {
    return (
      <div>
        {this.state.data.map((item) => (
          <p>{item}</p>
        ))}
      </div>
    )
  }
}
```

Zum Code →

CODE-BLOCK 3

Im Code-Block 3 ist das schematisch dargestellt. Um das gleiche Ergebnis mit Funktionskomponenten zu erreichen, wird *useEffect* eingesetzt. Dieser Hook nimmt als erstes Argument eine Funktion entgegen, die den auszuführenden Effekt darstellt. In unserem Beispiel würden wir genau hier unseren Netzwerk-Request starten und mittels Callback auf das Ergebnis des Requests lauschen. Den Callback nutzen wir ebenfalls dazu, den lokalen State zu aktualisieren. Da wir in einer Funktionskomponente sind, ist dieser lokale State natürlich mittels *useState* realisiert. Der Code-Block 4 zeigt die Netzwerk-Request-Variante mit Hooks.

```jsx
import React, { useEffect, useState } from "react"

export const LoadDataFunction = () => {
  const [data, setData] = useState()

  useEffect(() => {
    fetch("/some-data.json")
      .then((response) => response.json())
      .then((data) => {
        setData(data)
      })
  })

  return (
    <div>
      {data.map((item) => (
        <p>{item}</p>
      ))}
    </div>
  )
}
```

Zum Code →

CODE-BLOCK 4

Wann und wie oft die Effekt-Funktion ausgeführt wird, hängt vom zweiten Argument, welche *useEffect* übergeben wird, ab. Dies ist ein Array von Variablen, von denen die Effekt-Funktion abhängig ist. React beobachtet die Werte im Array und führt die Effekt-Funktion nur aus, wenn sich mindestens einer davon seit dem letzten Mal geändert hat. Üblicherweise handelt es sich bei den Variablen entweder um Props oder State-Variablen von *useState*-Hooks. Wenn beispielsweise der Netzwerk-Request von einem Eingabeparameter des Nutzers abhängt, sollte dieser Parameter im Array von *useEffect* auftauchen.

Dabei gibt es zwei Sonderfälle: Wird ein leeres Array übergeben, wird die Effekt-Funktion einmalig nach dem Einhängen der Komponente ausgeführt, vergleichbar mit *componentDidMount*. Wird der zweite Parameter dagegen komplett weggelassen, führt React die Effekt-Funktion bei jedem Rendering erneut aus.

Zurück zum Beispiel: Aktuell steckt in beiden Code-Beispielen noch ein Bug: Wird die Komponente aus dem Komponenten-Baum entfernt, bevor der Netzwerk-Request abgeschlossen ist (beispielsweise, weil der Nutzer sofort irgendwo anders hin navigiert), dann existiert die Komponenten-Instanz möglicherweise nicht mehr, wenn der Callback ausgeführt wird. Als Ergebnis erhalten wir Null-Pointer-Exceptions. Um dies bei der Klassen-Variante zu lösen, wird ein weiterer Lifecycle namens *componentWillUnmount* benötigt, in dem alle Aufräumarbeiten stattfinden können.

Bei der Hooks-Variante ist diese Situation bereits direkt im *useEffect*-Hook vorgesehen: Die Effekt-Funktion kann ihrerseits wiederum eine Funktion zurückgeben, die von React einmalig ausgeführt wird, sobald die Komponente entfernt wird. Der Code dazu ist im Code-Block 5 zu sehen.

```
1  useEffect(() => {
2    const controller = new AbortController()
3
4    fetch("/some-data.json", { signal: controller.signal })
5      .then((response) => response.json())
6      .then((data) => {
7        setData(data)
8      })
9
10   return () => {
11     controller.abort()
12   }
13 })
```

Zum Code →

CODE-BLOCK 5

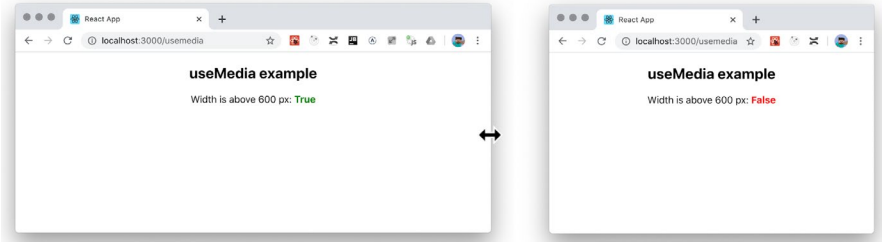
Man erkennt hieraus bereits einen Unterschied von *useState* im Vergleich zu den Lifecycle-Methoden bei Klassenkomponenten: Lifecycles folgen einem Gedankenmodell, welches sich stark auf die Komponente selbst bezieht, während der *useEffect*-Hook an einem häufig anzutreffenden Usecase ausgerichtet ist, nämlich das Ausführen eines bestimmten Seiteneffekts in Abhängigkeit von bestimmten Variablen. Um dies mit Lifecycles sauber zu implementieren, sind meist mehrere solcher Lifecycle-Methoden notwendig: Hängt der Request vom lokalen Zustand oder von Props ab, muss zusätzlich zu *componentDidMount* auch noch *componentDidUpdate* implementiert werden, sowie für Aufräumarbeiten *componentWillUnmount*. Während bei *useEffect* die gesamte Logik, inklusive Aufräumarbeiten, kompakt an einer einzigen Stelle zusammengefasst und bei Bedarf sogar wiederverwendbar als Custom-Hook ausgelagert werden kann, ist die Logik bei Klassenmethoden über mehrere Methoden verstreut.

Eigene Hooks bauen

Neben *useState* und *useEffect* bringt React noch eine Reihe weiterer Hooks mit, die für verschiedene Anwendungsfälle gedacht sind, beispielsweise *useRef*, um sich Zugriff auf DOM-Elemente zu verschaffen, oder *useContext*, um die Context-API von React zu nutzen. Spannend ist aber auch die Möglichkeit, auf Basis der existierenden Hooks seine eigenen Custom-Hooks zu bauen. Auf diese Weise können häufig verwendete Funktionalitäten gekapselt und zur Wiederverwendung bereitgestellt werden.

Viele Dritt-Bibliotheken aus der React-Community wie Redux oder die GraphQL-Library Apollo-Client bieten bereits Varianten mit Hooks an. Daneben existieren auch Sammlungen von nützlichen Hooks im Internet, an denen man sich bedienen oder inspirieren lassen kann.

Am Beispiel eines „useMedia“-Hooks wollen wir uns anschauen, wie man eigene Hooks entwickeln kann. Der *useMedia*-Hook soll einen Zugriff auf MediaQuery-Abfragen bieten, die sonst vor allem beim Styling mit CSS eine Rolle spielen. Damit lässt sich zum Beispiel prüfen, ob eine bestimmte Mindest-Breite unterschritten ist, um daraufhin das Layout für mobile Geräte anzupassen. Ein anderes Beispiel ist das Abfragen, ob der Anwender ein dunkles Farbschema bevorzugt. Die Query zur Abfrage des Darkmodes lautet *(prefers-color-scheme: dark)*, für die Mindestbreite wird z. B. *(min-width: 600px)* genutzt.



Unser Custom-Hook soll eine solche Abfrage als String entgegennehmen und als Ergebnis einen Boolean-Wert liefern, der angibt, ob die Abfrage zutrifft oder nicht. Die Browser-API für MediaQueries erlaubt nicht nur die Abfrage des aktuellen Zustands, sondern auch das Hinterlegen eines Listeners, der bei Änderung des Zustands feuert. Daher soll unser Hook ebenfalls dafür sorgen, dass bei einer Änderung des Abfrage-Ergebnisses die Komponente mit dem aktualisierten Wert neu gezeichnet wird. Dadurch können wir beispielsweise direkt darauf reagieren, wenn die Nutzer am Browser-Fenster ziehen und eine bestimmte Schwelle unter- oder überschreiten.

Der Code dazu, sowie ein Anwendungsbeispiel, ist in Code-Block 6 zu sehen. Zunächst benötigen wir *useState*, um den Zustand (matched die Query oder nicht?) abzubilden. Die Abfrage des Queries führen wir innerhalb von *useEffect* durch. Die Browser-API *window.matchMedia* liefert ein Query-Objekt, welches wir zunächst nutzen, um den aktuellen Zustand in unsere lokale Zustandsvariable zu übertragen. Anschließend hinterlegen wir einen Listener auf dem Query-Objekt, der bei Änderungen des Query-Ergebnisses ausgeführt wird. In diesem Fall aktualisieren wir ebenfalls unsere lokale Zustandsvariable.

Zum Schluss geben wir eine Aufräum-Funktion an *useEffect* zurück, die dafür sorgt, dass der Listener wieder korrekt abgebaut wird.

Indem wir den Eingabe-String des Nutzers als Abhängigkeit zum zweiten Parameter von *useEffect* hinzufügen, sorgen wir außerdem dafür, dass die MediaQuery-Abfrage erneut ausgeführt wird, sollte sich die Abfrage des Nutzers ändern.

```tsx
import React, { useEffect, useState } from "react"

const useMedia = (query: string): boolean => {
  const [matches, setMatches] = useState(false)

  useEffect(() => {
    const queryObject = window.matchMedia(query)
    setMatches(queryObject.matches)

    const listener = (e: MediaQueryListEvent) => {
      setMatches(e.matches)
    }

    queryObject.addEventListener("change", listener)

    return () => {
      queryObject.removeEventListener("change", listener)
    }
  }, [query])

  return matches
}

export const UseMediaExample = () => {
  const matches = useMedia("(min-width: 600px)")

  return (
    <>
      <p>Width is above 600 px: {matches ? "true" : "false"}</p>
    </>
  )
}
```

Zum Code →

CODE-BLOCK 6

Fazit

React-Hooks zählen sicherlich zu den tiefgreifendsten Erweiterungen, die React in den letzten Jahren erfahren hat. Nicht nur, weil damit Funktionskomponenten massiv aufgewertet wurden, sondern auch, weil mit Hooks viele Aufgaben wesentlich angenehmer, kompakter und besser verständlich umgesetzt werden können. Daher ist es nicht verwunderlich, dass die Community mit Freuden auf das neue Feature aufgesprungen ist.

Nicht nur stellen viele Dritt-Bibliotheken ihre Funktionalität, da wo es Sinn ergibt, nun auch als Hooks zur Verfügung. Hooks inspirieren Library-AutorInnen auch dazu, ganz neue Wege zu bestreiten und Bibliotheken zu entwickeln, die ohne Hooks wohl kaum denkbar wären. Als Beispiel dafür sei die State-Management-Bibliothek Recoil genannt, die auf der React Europe 2020 Konferenz vorgestellte wurde.

Spannend ist darüber hinaus, dass Hooks auch über die Grenzen der React-Community hinaus EntwicklerInnen inspirieren. So implementiert die Bibliothek Haunted die Hooks-API für Web Components und erlaubt damit das Erstellen von Custom-Components nach ähnlichen Mustern wie bei React-Funktionskomponenten. Auch die Vue.js-Entwickler arbeiten an einer API-Variante, die stark von React-Hooks inspiriert ist.

Für EntwicklerInnen von React-Anwendungen stellt sich abschließend die Frage, ob die eigene Code-Basis nun gänzlich von Klassenkomponenten befreit und alles auf Hooks umgebaut werden sollte. Auch wenn Funktionskomponenten und Hooks einige Vorteile haben, bleibt auch die Klassen-API weiterhin fester Bestandteil von React, weshalb es keinen wirklichen Grund gibt, bestehende und funktionierende Komponenten umzubauen. Bei der Entwicklung von neuen Komponenten dürfte es aber eine gute Idee sein, sich eher auf Funktionskomponenten zu konzentrieren.

Allerdings gibt es auch einige Usecases, für die bisher noch keine Hooks-Alternative zu Klassenkomponenten existiert. Dazu zählt zum Beispiel die Möglichkeit, sogenannte Error-Boundaries zu implementieren, die dafür sorgen, dass bei einem Fehler in einer Komponente nicht gleich die gesamte Anwendung abschmiert. Für die dafür notwendigen Lifecycle-Methoden *getDerivedStateFromError* und *componentDidCatch* existieren zum gegenwärtigen Zeitpunkt noch keine Hooks-Alternativen. Langfristig dürften aber auch diese Lücken geschlossen werden, so dass die Zukunft von React deutlich in Richtung Funktionskomponenten mit Hooks zeigt.

Der Autor

Nils Hartmann

Nils Hartmann ist freiberuflicher Softwareentwickler, Trainer und Coach aus Hamburg. Er programmiert in Java und JavaScript/TypeScript und unterstützt Teams mit Trainings und Beratung beim Ein- und Umstieg in die Entwicklung von Single-Page-Anwendungen sowie bei der Arbeit mit React, TypeScript und GraphQL.

Nils ist Autor des Buches „React – Grundlagen, fortgeschrittene Techniken und Praxistipps" (dpunkt, 2019).

React mit TypeScript entwickeln

React-Anwendungen können nicht nur in JavaScript, sondern auch in TypeScript entwickelt werden. TypeScript baut auf JavaScript auf und ergänzt die Sprache um ein statisches Typsystem. Dadurch sollen typische Probleme, die mit dem dynamischem Typsystem von JavaScript entstehen, vermieden werden. Dazu gehören insbesondere die bessere Wartbarkeit von Code in größeren und langlebigen Anwendungen.

Bevor wir uns nun ansehen, wie React-Komponenten typsicher mit TypeScript gebaut werden können, werfen wir zunächst einen Blick auf das Typsystem von JavaScript, um die Motivation hinter TypeScript zu verstehen, sowie auf die Grundlagen von TypeScript selbst.

Der folgende Code zeigt die dynamische Natur des Typsystems von JavaScript. Er ist syntaktisch gültig und könnte ohne Fehler im Browser ausgeführt werden.

```
let person = "Susi";
console.log(typeof person); // Ausgabe: string
console.log(person.toUpperCase()); // Ausgabe: SUSI

person = 32;
console.log(typeof person); // Ausgabe: number
console.log(person + 1); // Ausgabe: 33

person = function() { return "Kate" }
console.log(typeof person); // Ausgabe: function
console.log(person()); // Ausgabe: Kate
```

Zum Code →

Mit dem *typeof*-Operator wird der Typ, den die Variable *person* zu einem Zeitpunkt hat, auf der Konsole ausgegeben. Wir sehen, dass der Typ der Variable sich automatisch durch die Zuweisung eines Werts anderen Typs ändert.

Während die dynamische Typisierung auf der einen Seite sehr praktisch ist, weil wir uns keine Gedanken um die Typen machen müssen, kann sie auf der anderen Seite fehleranfällig und schwer verständlich sein. In dem kleinen Code-Beispiel oben können wir noch relativ einfach feststellen, von welchem Typ *person* ist. Spätestens bei größeren Anwendungen wird das aber schwieriger und damit fehlerträchtiger. Eine Hilfe für das Problem durch einen Compiler oder Typ-Checker gibt es nicht. Sehen wir uns noch ein zweites Beispiel an, diesmal eine Funktion:

```
function greet(person) { ... }
```

Wenn wir diese Funktion in unserer Anwendung aufrufen möchten, müssen wir wissen, was *person* sein soll: ein String? Oder ein Objekt? Und falls ja: welche Struktur hat das Objekt? Darf *person* auf *null* gesetzt sein, oder ist der Parameter ganz optional?

All dies können wir der Signatur nicht ansehen und sind auf die Dokumentation oder das Lesen des Source-Codes angewiesen. Auf der anderen Seite kann auch die Funktion selbst zur Laufzeit nicht sicher sein, dass ihr nur Parameter übergeben werden, die dem von ihr erwarteten Typ entsprechen. Nehmen wir beispielsweise an, die Funktion würde einen String erwarten, es wird ihr aber fälschlicherweise ein Objekt übergeben. Würde die Anwendung dann ausgeführt und die Funktion aufgerufen, wird erst zur Laufzeit ein „typischer" Fehler auftreten:

```
1  function greet(person) {
2    // toUpperCase ist auf einem String definiert, aber nicht auf einem Objekt
3    return person.toUpperCase();
4  }
5
6  greet({
7    name: "Susi"
8  })
9
10 // Laufzeitfehler: TypeError: p.toUpperCase is not a function
```

Zum Code →

Genau diese Probleme ergeben sich auch bei der Arbeit mit React-Komponenten, die in der Regel als Funktionen implementiert werden, welche als ersten Parameter Properties entgegennehmen können:

```
1  import React from "react";
2
3  function Hello(props) {
4    return <h1>Hello, {props.name}</h1>
5  }
```

Zum Code →

Wir wissen zwar, dass die React API vorschreibt, dass eine Komponente genau diesen einen Parameter für ihre Properties erwartet, können der Komponente aber nicht ansehen, ***welche*** Properties erwartet werden und von welchem Typ diese jeweils sind. Und ebensowenig wie im vorherigen JavaScript-Beispiel können wir auch innerhalb der Komponente nicht hundertprozentig sicher sein, dass wir die erwarteten Properties auch tatsächlich in der erwarteten Form übergeben bekommen. (Es gibt mit den React-PropTypes ein Modul für React, mit dem Properties beschrieben werden können, allerdings erfolgt auch hier die Prüfung lediglich zur Laufzeit und hilft nur sehr eingeschränkt während der Entwicklung).

TypeScript

Diese Probleme können – zumindest in großen Teilen – mit einem statischen Typsystem bereits zur Entwicklungszeit unterbunden werden. Prominente Sprachen mit statischem Typsystem sind etwa Java, C# oder C/C++.

Für JavaScript stellt TypeScript ein Typsystem zur Verfügung. Da TypeScript die Syntax von JavaScript übernimmt und lediglich erweitert (aber nicht verändert), ist jeder gültige JavaScript-Code auch gültiger TypeScript-Code. In der einfachsten Form ist die Umwandlung einer Datei mit JavaScript-Code auf TypeScript dadurch erfolgt, dass die Dateiendung von *.js* auf *.ts* geändert wird. (In einem TypeScript-fähigen IDE oder Editor kann man auch eine JavaScript-Datei von TypeScript überprüfen lassen, in dem man am Anfang der Datei *//@ts-check* einfügt).

Das erste, oben gezeigte Beispiel ist also auch gültiger TypeScript-Code. Allerdings würde der Typchecker von TypeScript bereits bei der zweiten Zuweisung eine Fehlermeldung ausgeben:

```
1  let person = "Susi";
2  console.log(typeof name); // Ausgabe: string
3  name = 32;   // TypeScript Fehlermeldung: Type '32' is not assignable to type 'string'
```

Zum Code

Obwohl der Code sich an dieser Stelle nicht von der JavaScript-Variante unterscheidet, hat TypeScript der Variablen *person* bereits (implizit) einen Typ zugewiesen, nämlich *string*. Dieses Verhalten nennt sich Type *Inference* und bedeutet, dass TypeScript Typen automatisch und selbstständig herleitet, wo sie nicht explizit angegeben wurden. In diesem Fall hat TypeScript für die Variable "person" den Typ *string* ermittelt und da einem String keine Zahl (Typ *number*) zugewiesen werden kann, gibt TypeScript in der folgenden Zeile eine entsprechende Fehlermeldung aus.

Neben der impliziten Herleitung von Typen ist es auch möglich (und an einigen Stellen auch erforderlich), Typangaben explizit hinzuschreiben. Parameter von Funktionen etwa müssen immer mit einer Typangabe

versehen werden, da TypeScript hier nicht in der Lage ist, die korrekten Typen selbst zu ermitteln. Typ-Angaben werden in TypeScript durch einen Doppelpunkt getrennt hinter eine Variable, ein Funktionsargument oder eine Funktionssignatur (als Rückgabewert der Funktion) geschrieben. Die *greet*-Funktion aus dem obigen Beispiel könnte in TypeScript nun so aussehen:

```
function greet(person: string) {
  return person.toUpperCase();
}

greet({
 name: "Susi"
});

// TypeScript Fehler: Argument of type '{ name: string; }'
// is not assignable to parameter of type 'string'.

const greeting = greet("Susi"); // OK

greeting.toLowerCase(); // OK, greeting ist string und darauf ist die Funktion toLowerCase definiert.
```

Zum Code →

TypeScript stellt hier bereits zur Entwicklungszeit sicher, dass die *greet*-Funktion nur mit einem String aufgerufen wird. Alle anderen Typen führen zu einem Fehler. Der Rückgabewert der Funktion wird wiederum von TypeScript automatisch hergeleitet, so dass die Variable *greeting* ebenfalls vom Typ *string* ist.

Neben den Datentypen wie *string*, *number*, *boolean* sind in TypeScript auch *null* oder *undefined* jeweils eigene Typen. Variablen, Funktionsargumente etc., die *null* oder *undefined* annehmen können, müssen aus diesem Grund auch extra gekennzeichnet werden. Der Aufruf von *greet* mit *null* als Argument (oder ganz ohne Angabe eines Arguments) würde mit einem Fehler von TypeScript quittiert werden. Um anzuzeigen, dass ein Typ auch *null* oder *undefined* sein kann, kann ein Typ verwendet wer-

den, der sich aus mehreren einzelnen Typen zusammensetzt (Union Type). Dazu werden einfach mehrere Typen durch den or-Operator (|) getrennt hingeschrieben. Die Signatur der *greet*-Funktion könnte wie folgt geändert werden, um auch null-Werte zu akzeptieren:

```
function greet(person: string | null) {
  return person.toUpperCase(); // TS Fehler: Object is possibly 'null'
}
```

Zum Code →

Da *person* nun zur Laufzeit auch *null* sein kann, gibt es beim Zugriff darauf eine entsprechende Fehlermeldung. Auf diese Weise verhindert TypeScript potentielle Fehlerquellen. Den entsprechenden Code könnten wir wie folgt korrigieren:

```
function greet(person: string | null) {
  if (person === null) {
    return "";
  }
  return person.toUpperCase(); // OK
}
```

Zum Code →

Hier weiß TypeScript nun, dass *person* in der letzten Zeile der Funktion nur noch vom Typ *string* sein kann, weil die andere Möglichkeit im *if*-Zweig davor überprüft wurde. Dieses Feature nennt sich „Type Narrowing“, also „Typ Eingrenzungen“: je nach Programmzweig kann sich der Typ einer Variablen von einer Menge an Typen auf eine Untermenge reduzieren.

Eigene Typen definieren

Neben den primitiven Typen ist es auch möglich, eigene Typen mit TypeScript zu definieren. Damit lassen sich Strukturen von Objekten beschreiben. Beispielsweise könnten wir für eine andere Variante der *greet*-Funktion ein *person*-Objekt beschreiben. TypeScript stellt dann sicher, dass nur Aufrufe an die Funktion erlaubt sind, deren Objekte der erforderlichen Struktur entsprechen. Ein eigener Typ kann mit dem Schlüsselwort *type* definiert werden. Alternativ kann auch das Schlüsselwort *interface* verwendet werden. Die Unterschiede zwischen *interface* und *type* sind marginal und können für die folgenden Beispiele vernachlässigt werden.

In der Definition werden sämtliche Properties (zu denen natürlich auch Funktionen gehören können) des Objektes samt ihrer Typen angeben:

```
1  type Person = { name: string, lastname: string };
2
3  function greet(person: Person) {
4    return `Hello, ${person.name} ${person.lastname}`
5  };
6
7  greet({name: "Susi", lastname: "Mueller"}); // OK
8  greet({name: "Klaus"}); // TS Error: Property 'lastname' missing
```

Zum Code →

React-Komponenten mit TypeScript

Das TypeScript Typsystem bietet noch sehr viel mehr Möglichkeiten, aber bereits mit den bis hierher gezeigten Features lassen sich React-Komponenten mit TypeScript entwickeln.

Um TypeScript in eigenen Projekten zu verwenden, muss das Projekt entsprechend konfiguriert sein. Dazu gehört, dass der TypeScript-Compiler verwendet wird und die TypeScript-Typ Deklarationen für React im Projekt eingebunden sind. Am einfachsten geht das, in dem man das Tool create-react-app verwendet, das neben eines JavaScript-basierten auch ein TypeScript-basiertes React Projekt aufsetzen kann. Dazu muss das Argument *--template typescript* verwendet werden:

```
npx create-react-app my-app --template typescript
```

In diesem Projekt kann nun ohne weitere Konfiguration sofort mit der Entwicklung von React-Komponenten in TypeScript begonnen werden. TypeScript unterstützt sogar die in React verwendete Spracherweiterung JSX. Dazu müssen Dateien, die JSX-Code enthalten, allerdings zwingend mit der Endung *.tsx* benannt werden.

Als Beispiel, wie eine React-Komponente mit TypeScript gebaut werden kann, sehen wir uns folgende einfache Komponente an, die ein Login-Formular mit zwei Input-Feldern und einem Button rendert:

```
function LoginForm(props) {
  const [username, setUsername] = React.useState("");
  const [password, setPassword] = React.useState("");

  return (
    <form>
      <h1>{props.title.toUpperCase()}</h1>

      <input
        type="text"
        value={username}
        onChange={(e) => setUsername(e.currentTarget.value)}
      />

      <input
        type="password"
        value={password}
        onChange={(e) => setPassword(e.currentTarget.value)}
      />
      <button onClick={() => props.onLoginClick(username, password)}>
        Login
      </button>
    </form>
  );
}
```

Zum Code →

Um die Komponente mit TypeScript zu verwenden, muss zunächst beschrieben werden, wie das *props*-Argument aussieht, in dem die übergebenen Properties einer Komponente enthalten sind.

Die *LoginForm*-Komponente erwartet zwei Properties: einmal den Titel für das Formular und eine Callback-Funktion (*onLoginClick*) die aufgerufen wird, wenn auf den Button geklickt wird. Der *onLoginClick*-Funktion

werden dann die Eingaben aus dem Formular (*username* und *password*) übergeben. Genau diese Signatur kann in dem Typ für das *props*-Objekt angegeben werden:

```
type LoginFormProps = {
  title: string;
  onLoginClick(username: string, password: string): void;
};

function LoginForm(pros: LoginFormProps) { ... }
```

Zum Code →

Innerhalb einer React-Komponente dürfen die eingereichten Properties nicht verändert werden. Diesem Umstand kann die Typ-Definition des Property-Typen Rechnung tragen, in dem dort die einzelnen Properties als *read-only* markiert werden. Das kann man tun, in dem man die Definition des Typs noch mit dem Hilfstyp *Readonly* umschließt. An der Verwendung des Typs ändert sich nichts.

```
type LoginFormProps = Readonly<{
  title: string;
  onLoginClick(username: string, password: string): void;
}>;
```

Zum Code →

An der Verwendung der Komponente ändert sich durch TypeScript auch nichts. Hier wird wie aus JavaScript-basierten React-Anwendungen die JSX-Notation verwendet, um die Komponente einzubinden.

```
...
<LoginForm
  title="Login"
  onLoginClick={
   (username, password) => console.log(username, password)
  }
/>

...
```

Zum Code →

Mit der Angabe der Typ-Information für die Properties der ***LoginForm***-Komponente kann TypeScript nun eine ganze Reihe von Dingen sicherstellen:

- Verwender der Komponente werden gezwungen, die Properties samt Callback-Funktion korrekt anzugeben. Wenn zum Beispiel das ***title***-Property nicht angegeben wird, erhält der Verwender der Komponente einen entsprechenden Fehler bereits während der Entwicklung oder später auch im CI-Build.

- Auch die Komponente kann sich demnach darauf verlassen, dass sie die korrekten Properties übergeben bekommen hat. Sie kann also sicher sein, dass zum Beispiel das ***title***-Property auf jeden Fall ein String ist (und nicht ***null*** oder ein anderer Datentyp).

- Die in TypeScript geschriebene Komponente kann auch aus JavaScript-Code aufgerufen werden. Somit steht sie nicht ausschließlich TypeScript-basierten Anwendungen zur Verfügung. Bei der Verwendung in

JavaScript-Code sind dann natürlich keine Typ-Checks mehr möglich und die geschilderten Probleme können auftreten.

- Innerhalb der Komponente können wir nicht (versehentlich) auf Properties zugreifen, die es gar nicht gibt. Ein Fehler in der Art *props.titel* statt *props.title* löst einen Fehler in TypeScript aus. Außerdem können Properties nicht versehentlich geändert werden (*props.title = "Moin moin!"*), da wir die Properties als *read-only* markiert haben.

- Sowohl der Verwender der Komponente als auch innerhalb der Komponente selbst kann eine TypeScript-kompatible IDE oder ein TypeScript-fähiger Editor Code Completion und andere Hilfen zur Verfügung stellen.

- Innerhalb der Komponente kennt TypeScript den Typ der Properties, weiß also, dass *title* ein String und nicht *null* ist (und somit darauf *toUpperCase()* aufgerufen werden kann). Da auch die Signatur der Callback-Funktion *onLoginClick* angegeben ist, kann TypeScript bei der Verwendung der Funktion sicherstellen, dass sie mit den korrekten Argumenten aufgerufen wird.

TypeScript Fehler bei fehlerhafter Verwendung der LoginForm-Komponente

Neben den Properties verwenden wir in der Komponente auch Zustand (State) und zwar für username und password. In der allereinfachsten Form brauchen wir dafür keine explizite Typ-Angabe hinzuschreiben, da TypeScript – aufgrund der Typ-Deklaration für die *useState*-Funktion – in der Lage ist, den Typ herzuleiten. Aus diesem Grund gibt es auch kein Problem beim Aufruf der *onLoginClick*-Callback-Funktion: Da TypeScript sowohl für *username* als auch *password* den korrekten Typ (*string*) ermittelt hat, weiß TypeScript folglich auch, dass der Aufruf korrekt ist.

Auf diese Weise sind auch die von *useState* zurückgelieferten Setter-Funktionen (*setUsername*, *setPassword*) korrekt getypt. Folgende Aufrufe dieser Funktionen würden nun zu Fehler führen:

```
setUsername(null); // TypeScript Fehler: Argument of type
‘null‘ is not assignable to parameter
setUsername(123); // TypeScript Fehler: Argument of type
‘123‘ is not assignable to parameter
```

Es gibt Fälle, in denen TypeScript den Typ für den Zustand nicht selbständig herleiten kann. Zum Beispiel, wenn der Zustand auch *null* sein darf oder aus einem komplexen Objekt besteht. In solchen Konstellationen kann auch dem *useState*-Hook ein Typ-Parameter übergeben werden, der den State beschreibt. Nehmen wir an, der State für den Username kann auch *null* annehmen, könnten wir schreiben:

```
const [username, setUsername] = React.useState<string|null>(“”);
```

Da *username* nun in der Komponente *null* sein kann, zwingt uns TypeScript bei der Verwendung dazu, entsprechende *null*-Prüfungen vor der Verwendung durchzuführen, damit es zur Laufzeit nicht zu Fehlern kommt.

Das vollständige Beispiel befindet sich in der Online IDE Codesandbox.

Ausblick

Wir haben gesehen, dass sich React-Komponenten nahtlos mit TypeScript entwickeln lassen. Im besten Fall müssen wir nicht einmal Typ-Definitionen explizit schreiben, profitieren aber schon von TypeScripts statischem Typsystem. Der TypeScript-Support auch in den weiteren React APIs ist sehr gut. Überall ist es möglich, eigene Typen anzugeben. So kann man zum Beispiel beschreiben, wie ein Objekt aussieht, dass mit der React Context API der Anwendung zur Verfügung gestellt wird. Modifikation und Zugriff auf das globale Objekt sind dann typsicher.

Für den *useReducer*-Hook kann sowohl sichergestellt werden, dass der in der zugehörigen Reducer-Funktion verwaltete Zustand korrekt verwendet wird (beim Verändern innerhalb des Reducers und beim Verwenden innerhalb der Komponente) und dass die Komponente auch nur bekannte und korrekte Actions mittels der *dispatch*-Funktion auslöst.

Auch große Teile des React-Ökosystems kommen mit TypeScript klar (und sind teilweise sogar in TypeScript implementiert). Testfälle mit Jest und der react-testing-library lassen sich out-of-the-box in TypeScript schreiben und auch der React Router und Redux funktionieren mit TypeScript. Insbesondere in Redux kann TypeScript eine große Hilfe sein, um sicherzustellen, dass Actions, Reducer und Komponenten korrekt zusammenspielen. Dank der Type Inference ist es dabei an vielen Stellen nicht einmal nötig, explizit Typdefinitionen zu schreiben.

Foto: Sebastian Haase

Die Autorin

Lisa Oppermann

Lisa Oppermann ist selbstständige JavaScript-Entwicklerin mit Schwerpunkt auf React und React-Native. Sie arbeitet für verschiedene Kunden von der Fin-Tech bis zur Smart-Mobility Branche und hat unter anderem in diversen Applikationen Redux eingeführt.

Redux: State-Management für React

Mit React ist es möglich, komplexe Benutzeroberflächen zu bauen. Webapplikationen wie Facebook, Airbnb und Netflix benutzen ebenso React wie auch Host Europes Website, von der Sie dieses E-Book heruntergeladen haben.

Je aufwendiger eine Applikation und je mehr Daten gleichzeitig verarbeitet werden, desto anspruchsvoller wird es, die Applikationsdaten zu verwalten und den Überblick über den aktuellen Zustand der Anwendung zu behalten. Wenn man diesen Zustand der Applikation organisiert, spricht man vom ***State-Management***.

Der State einer Applikation kann sich von einem kleinen Detail bis hin zum Gesamtstatus der Applikation erstrecken. Im Prinzip kann alles als State betrachtet werden: Ist der Benutzer eingeloggt, welche Daten werden gerade angezeigt, ist das Menü auf- oder zugeklappt, wurde Text in ein Textfeld eingegeben, hat der Benutzer bereits das Cookie-Modal gesehen usw.

Zusammenfassend stellt sich beim Schreiben einer komplexen Applikation die Frage: In welchem ***Zustand*** oder ***Status*** befindet sich jetzt gerade die Applikation, und wie kann man diesen Status abbilden und Änderungen nachvollziehen?

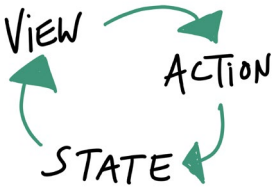
Vom lokalen zum globalen State-Management

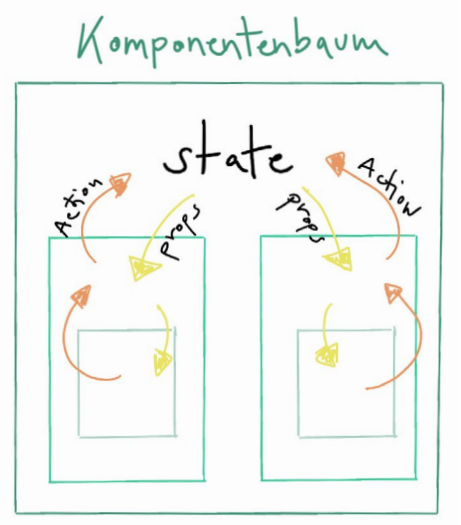
Wir wissen bereits aus vorangegangenen Kapiteln, dass eine React-Komponente selber einen lokalen State speichern kann. Wenn ein Komponenten-State Daten enthält, die von anderen Komponenten

genutzt werden sollen, werden diese über Props an verschachtelte Unterkomponenten weitergegeben. In der Praxis bedeutet dies meist, dass die oberste Komponente innerhalb des Komponentenbaums den Zugriff auf veränderbare Daten im lokalen Komponenten-State hält und die Werte weitergibt. Wenn ein State-Wert durch Unterkomponenten geändert werden kann, gibt die oberste Komponente einen Action-Callback als Prop weiter, und die Unterkomponente kann damit den State der obersten Komponente ändern, die diese Änderungen wiederum zurück an die Unterkomponenten gibt.

Dies ist der übliche Datenfluss von React: Eine Aktion oder auch Event verändert den Zustand einer Komponente, die daraufhin den neuen Wert als Props an andere Komponenten weitergibt. Dies ist ein uni-direktionaler Datenaustausch, da – wie der Name schon sagt – die Daten nur in eine Richtung weitergegeben werden.

Uni-direktionaler Datenaustausch



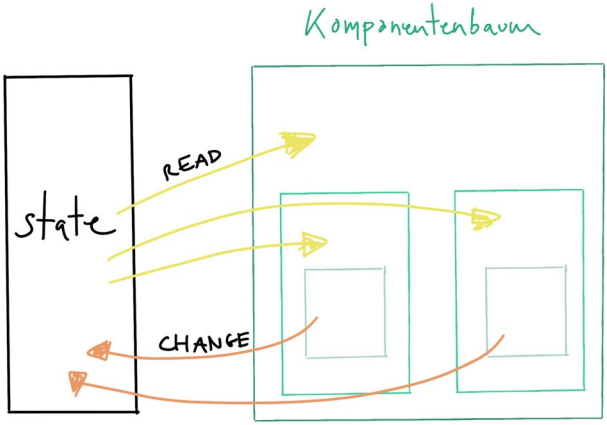


Je komplexer aber die Datenstrukturen werden und je differenzierter das User Interface sein soll, das diese Daten anzeigt, desto aufwendiger wird es, diese Daten durch mehrere Komponentenebenen durchzureichen.

Für kleine, überschaubare Applikationen mag das zu verschmerzen sein, will man aber eine komplexe Oberfläche darstellen, deren einzelne Teilbereiche mit unterschiedlichen Datenquellen gerendert werden, kommt die Frage auf: Geht das nicht einfacher?

Hier kommt das ***globale State-Management*** ins Spiel, das den Zustand bzw. die notwendigen Daten einer Applikation unabhängig vom lokalen State der Komponenten in einer Art Container hält und den Komponenten zur Verfügung stellt.

Die Daten werden demnach nicht mehr durch einzelne Komponentenebenen durchgereicht, sondern eine tief verschachtelte Komponente kann sich direkt mit dem globalen State verbinden und hat Zugriff auf dessen Daten. Dies vereinfacht nicht nur den Datenfluss, sondern verhilft zu einer weit besseren Übersicht über den aktuellen Zustand der Applikation:



Was ist Redux?

Redux ist eine JavaScript-Bibliothek, die ein Konzept bereitstellt, durch das es möglich ist, den State einer Applikation zu lesen, zu ändern und abzubilden. Mit Redux wird ein Zustands-Container (***Store***) erstellt, der den globalen Applikations-Zustand verwaltet (***State***). Weiterhin hält der Store Methoden bereit, durch die der Zugriff zum State bereitgestellt wird und mit denen Daten im State verändert werden können:

- ***getState*** => Daten im State werden gelesen
- ***dispatch*** => Daten im State werden verändert
- ***subscribe*** => Callback, sobald der State geändert wurde

State

Der globale State wird in einem simplen Objektbaum innerhalb des Stores gespeichert und kann mit der Methode ***getState*** gelesen werden. Pro Applikation gibt es nur einen State, der als Basis dafür gilt, was die Applikation darstellt und wie sie sich verhält. Der State ist demnach die "***Single Source of Truth***" – die einzige Quelle der Wahrheit. Damit dies so bleibt, wird der State selber nie direkt manipuliert, sondern nur lesend zur Verfügung gestellt.

Action

Die einzige Möglichkeit, den State zu ändern, besteht darin, eine ***Aktion*** (action) mit der Methode ***dispatch*** auszulösen. Eine Action ist ein einfaches Objekt, das beschreibt, was sich ändern soll. Dieses Action-Objekt muss das Feld "***type***" enthalten, dessen Wert ein String ist und das den Namen der Action darstellt. Darüber hinaus kann das Action-Objekt individuell ausgebaut werden, um weitere Daten in der Action weiterzugeben. Dafür werden ***Action Creators*** geschrieben. Dies sind simple Funktionen, deren Rückgabewert ein Action-Objekt ist:

```
// Ein Action Objekt
const simpleAction = {
  type: "TOGGLE_MAIN_MENU"
};
dispatch(simpleAction);


// Ein Action Objekt mit Payload / Daten
const simpleActionWithPayload = {
  type: "SET_USER_NAME",
  name: "JANE DOE"
};
dispatch(simpleActionWithPayload);


// Action Creators = Funktionen, die ein Action Objekt zurückgeben
const setIncrement = value => ({
  type: "INCREMENT",
  value
});
dispatch(setIncrement(2));

const setUserName = name => ({
  type: "SET_USER_NAME",
  name
});
dispatch(setUserName("JANE DOE"));
```

Zum Code →

Reducer

Actions werden durch ***Reducer*** verarbeitet, die darüber entscheiden, wie sich der State der Applikation ändern soll, nachdem mit ***dispatch*** eine Action an den Store geschickt wurde. Reducer sorgen dafür, dass das

existierende State-Objekt nicht direkt verändert, sondern mit der Action ein neuer State erstellt wird.

Klassischerweise werden Reducer als switch-Statement geschrieben, jedoch kann man auch mit simplen if-Statements denselben Effekt erreichen oder Hilfswerkzeuge wie Redux Toolkit benutzen. Alle Möglichkeiten haben aber eins gemeinsam: Reducer verändern den State nicht direkt, sondern erstellen immer einen neuen State, der zurückgegeben wird. Reducer sind außerdem ***pure functions***, die nicht auf externe Variablen zugreifen oder Seiteneffekte aufrufen und immer denselben Wert zurückgeben (siehe auch Wikipedia zu pure functions).

```
// Reducer sind pure functions
// Reducer nehmen den alten State und eine Action als Parameter
// Durch die Aktion wird ein neuer State zurückgegeben
// Oder der alte State, wenn kein passender Action Type definiert wurde

const reducer = (state = 0, action) => {
  switch (action.type) {
    case 'INCREMENT': {
      return state + action.value;
    }
    case 'DECREMENT': {
      return state - action.value;
    }
    default: {
      return state; // return default state
    }
  }
};
```

Zum Code →

Store

Sobald *Actions* definiert und ein Reducer erstellt wurden, kann man den Redux Store initialisieren. Dafür stellt Redux die Methode *createStore* zur Verfügung, der man bei Aufruf einen Reducer als Argument mitgibt. Die Methode gibt daraufhin den Store zurück, der – wie oben gezeigt – die drei Methoden *getState*, *dispatch* und *subscribe* bereithält.

```
import { createStore } from 'redux'

// Reducer: eine "pure function" mit (state, action) => state Signatur.
// Reducer entscheiden, wie ein neuer State erstellt wird.
const counterReducer = (state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + action.value
    case 'DECREMENT':
      return state - action.value
    default:
      return state
  }
}

// Einen Redux Store erstellen, der die Methoden getState, dispatch und subscribe zur Verfügung stellt
const store = createStore(counterReducer)

// Mit subscribe einen Callback definieren sobald der State sich ändert
store.subscribe(() => console.log(store.getState()))
// => 0

// Mit dispatch Actions auslösen, die Informationen über die State-Änderungen enthalten
// und vom Reducer verarbeitet werden
store.dispatch({ type: 'INCREMENT', value: 10 })
// => 10
store.dispatch({ type: 'DECREMENT', value: 2 })
// => 8
```

Zum Code →

Letztendlich ist der Redux Store nur ein Objekt mit ein paar Methoden. Und mit *dispatch* werden weitere Objekte über die Reducer an den Store gegeben (Objekt mit type und *payload*), durch die im Reducer ein neuer State erstellt wird. Wir schieben also eigentlich nur simple Objekte mit synchronen Funktionen hin und her. Das ist fast alles.

Asynchrone Actions mit Middleware

Bei asynchronen Datenflüssen ist die Handhabung mit Redux allerdings ein bisschen aufwendiger.

Die oben beschriebenen Action Creators und Reducer beziehen sich auf eine synchrone Datenverarbeitung, die direkt auf dem Client (Browser) ausgeführt wird, ohne dass Netzwerk-Requests geschickt werden. In den meisten produktiven Applikationen werden jedoch Daten verarbeitet, die von einem Server kommen, indem Netzwerk-Requests zum Abrufen und Speichern von Daten zwischen Client und Server verschickt werden.

Ein Redux Store weiß nichts über asynchrone Datenflüsse und arbeitet jede Action durch die Reducer der Reihe nach ab, so wie sie an den Store mit ***dispatch*** geschickt wurden. Das bedeutet, dass asynchrone Datenverarbeitungen außerhalb des Stores erfolgen müssen. Dafür hat Redux das Konzept der ***Middleware*** erstellt, durch das es möglich ist, den Store um weitere Funktionen zu erweitern.

Eine ***Middleware*** ist das Zwischenstück zwischen Action und Reducer. Durch eine Middleware ist es möglich, eine Action zu ändern, zu erweitern, zu stoppen oder zu verzögern, ***bevor*** diese den Reducer erreicht. Wenn man also zum Beispiel Netzwerk-Requests an den Server schickt, will man zunächst das Ergebnis abwarten und danach entscheiden, ob der Redux-State durch eine Action aktualisiert werden muss. Da dies ein häufiger Anwendungsfall ist, stellt Redux die Middleware Redux-Thunk zur Verfügung, die für diesen Anwendungsfall genutzt werden kann und direkt beim Initialisieren des Stores mitgeben wird:

```
1  import { createStore, applyMiddleware } from 'redux';
2  import thunk from 'redux-thunk';
3  import rootReducer from './reducers';
4
5  const store = createStore(rootReducer, applyMiddleware(thunk));
```

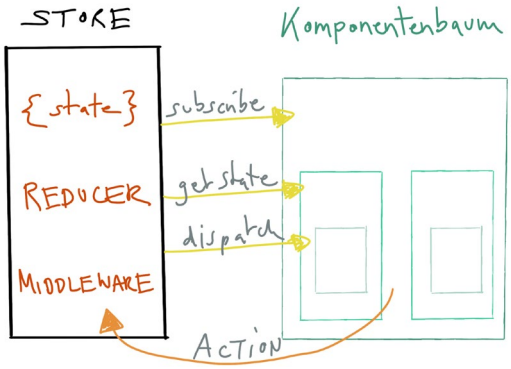
Zum Code →

Redux-Thunk leistet Folgendes: Wird eine Action an den Store dispatched, die ein simples Action-Objekt zurückgibt, dann wird diese Action direkt an den Reducer weitergeleitet. Wird allerdings eine Action dispatched, die kein Objekt zurückgibt, sondern ***eine weitere Funktion***, dann wird diese zurückgegebene Funktion mit den beiden Redux-Store-Methoden ***getState*** und ***dispatch*** aufgerufen. Dadurch ist es möglich, asynchrone Actions oder auch Thunk-Actions auszuführen, die Zugriff auf den State und die Methode ***dispatch*** haben.

```
// Asynchrone Action oder auch Thunk Action
// die durch die Middleware Redux-Thunk mit dispatch und getState ausgeführt wird
// und somit innerhalb der Action auf diese Methoden zugegriffen werden kann

export const fetchUser = () => {
  return async (dispatch, getState) => {
    const state = getState(); // read the current state

    try {
      const response = await fetch('some-url');
      const data = await result.json();

      if (state.user.first_name !== data.first_name) {
        dispatch(setFirstName(data.first_name)); // dispatch a simple Action Creator
      }
    } catch (error) {
      dispatch(handleError(error)); // a thunk action can call another thunk action
    }
  };
};

dispatch(fetchUser());
```

Zum Code →

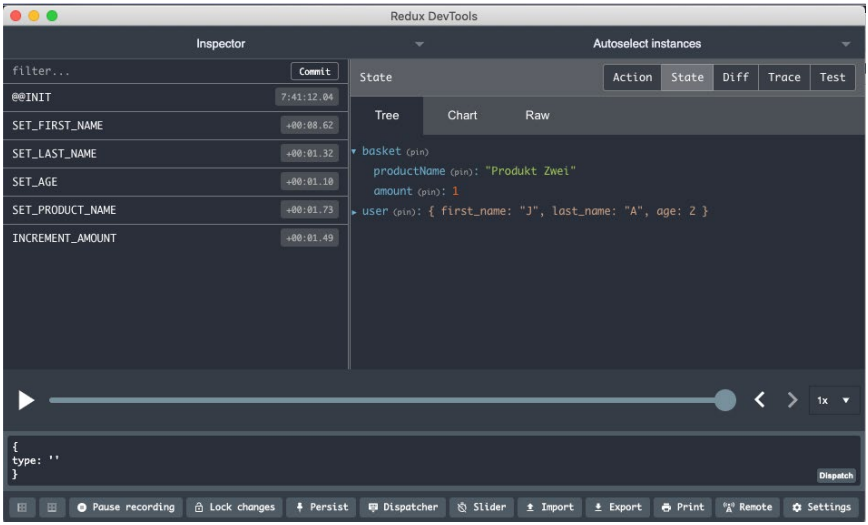
Damit haben wir die fundamentalen Konzepte von Redux kennengelernt: Store, State, Actions, Reducer, Middleware und asynchrone Actions.



Redux DevTools

Um zu visualisieren, wie der State aussieht und welche Actions an den Store geschickt wurden, kann man die Redux DevTools installieren. Die dazugehörige Browser-Erweiterung hilft beim Debugging und State-Monitoring und ermöglicht es, die State-Veränderungen wieder rückgängig zu machen (Time-Travel). Wie die DevTools installiert werden, wird auf der offiziellen Website erklärt.

Hier eine Beispielansicht der DevTools Browser-Erweiterung:



Redux Toolkit

Redux empfiehlt vor allem Anfängern, die Bibliothek Redux Toolkit zu benutzen, wenn Redux in einer Applikation eingebunden werden soll. Redux Toolkit stellt diverse Funktionen zur Verfügung, die das Schreiben von Actions und Reducer sowie das Erstellen eines Stores vereinfachen. Ohne theoretischen Hintergrund allerdings ist auch Redux Toolkit nicht sofort zugänglich, weshalb er in diesem Artikel nicht behandelt wird.

Installation

Um Redux in einer React-Applikation zu benutzen, werden folgende Node-Pakete benötigt, die über die Kommandozeile installiert werden:

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

oder

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

Redux-Thunk wird als Middleware installiert, damit die Applikation mit asynchronen Actions arbeiten kann.

Die Einbindung von Redux kann in den verschiedensten JavaScript-Applikationen erfolgen und beschränkt sich nicht nur auf die Nutzung von React. Deswegen gibt es für die Implementierung von Redux in eine React-Applikation eine offizielle React-Bindung: React Redux. Diese Bibliothek enthält React-Komponenten, die in einer React-Applikation benutzt werden können, um dadurch die Verwendung von Redux deutlich zu vereinfachen.

Erstellung von State-Struktur und Store

Sobald die Pakete installiert sind, sollte man sich Gedanken über die State-Struktur machen. Wie soll der State-Objektbaum aufgebaut werden, damit die Datenstruktur der Applikation sinnvoll aufgeteilt ist?

Um die Beispiele aus den vorherigen Abschnitten zu erweitern, könnte ein einfacher State-Shape so aussehen:

```
{
  user: {
    firstName: '',
    lastName: '',
    age: ''
  },
  basket: {
    productName: '',
    amount: 0
  }
}
```

Dieser State ist sehr überschaubar und soll hier lediglich als Beispiel dienen. In einer produktiven Applikation wäre ein Redux State weitaus umfangreicher.

Wir bauen also eine einfache Applikation, bei der User-Daten angelegt und in einem Einkaufswagen (basket) ein Produkt und dessen Anzahl gespeichert werden können. Hierfür werden wir zwei Reducer erstellen: *userReducer* und *basketReducer*, die jeweils für den dazugehörigen Teil des States zuständig sein sind.

Der UserReducer und Actions:

```
// App/store/user/reducer.js

import {
  SET_FIRST_NAME,
  SET_LAST_NAME,
  SET_AGE
} from 'App/store/user/actions';

const initialState = {
  firstName: '',
  lastName: '',
  age: ''
};

export default (state = initialState, action) => {
  switch (action.type) {
    case SET_FIRST_NAME: {
      return {
        ...state,
        firstName: action.firstName,
      };
    }
    case SET_LAST_NAME: {
      return {
       ...state,
       lastName: action.lastName,
      };
    }
    case SET_AGE: {
      return {
       ...state,
       age: action.age,
      };
    }
    default: {
      return state;
    }
  }
};
```

Zum Code →

REDUCER

```
// App/store/user/actions.js

export const SET_FIRST_NAME = 'SET_FIRST_NAME';
export const SET_LAST_NAME = 'SET_LAST_NAME';
export const SET_AGE = 'SET_AGE';


// action creators which return plain action objects

export const setFirstName = firstName => ({
  type: SET_FIRST_NAME,
  firstName
});

export const setLastName = lastName => ({
  type: SET_LAST_NAME,
  lastName
});

export const setAge = age => ({
  type: SET_AGE,
  age
});
```

Zum Code →

ACTIONS

Der BasketReducer und Actions:

```
// App/store/basket/reducer.js

import {
  SET_PRODUCT_NAME,
  INCREMENT_AMOUNT,
  DECREMENT_AMOUNT
} from 'App/store/basket/actions';


// Der initiale State ohne Werte
const initialState = {
  productName: '',
  amount: 0
};

export default (state = initialState, action) => {
  switch (action.type) {
    case SET_PRODUCT_NAME: {
      return {
        ...state,
        productName: action.name
      };
    }
    case INCREMENT_AMOUNT: {
      return {
        ...state,
        amount: state.amount + 1
      };
    }
    case DECREMENT_AMOUNT: {
      return {
        ...state,
        amount: state.amount - 1
      };
    }
    default: {
      return state;
    }
  }
};
```

Zum Code →

REDUCER

```
// App/store/basket/actions.js

export const SET_PRODUCT_NAME = 'SET_PRODUCT_NAME';
export const INCREMENT_AMOUNT = 'INCREMENT_AMOUNT';
export const DECREMENT_AMOUNT = 'DECREMENT_AMOUNT';


// Action Creators die Action Objekte zurückgeben

export const setProductName = name => ({
  type: SET_PRODUCT_NAME,
  name
});

export const incrementAmount = () => ({
  type: INCREMENT_AMOUNT
});

export const decrementAmount = () => ({
  type: DECREMENT_AMOUNT
});
```

Zum Code →

ACTIONS

Mit der Methode *combineReducers* von Redux ist es möglich, mehrere Reducer für verschiedene State-Objekt-Trees zuzuweisen. Ist dies getan, kann der Redux Store mit der Methode *createStore* erstellt werden:

```
// App/store/index.js

import { combineReducers, createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';

import userReducer from 'App/store/user/reducer';
import basketReducer from 'App/store/basket/reducer';

const rootReducer = combineReducers({
  user: userReducer,
  basket: basketReducer
});

export const store = createStore(rootReducer, applyMiddleware(thunk));
```

Zum Code →

Den Store an React-Komponenten weiterreichen

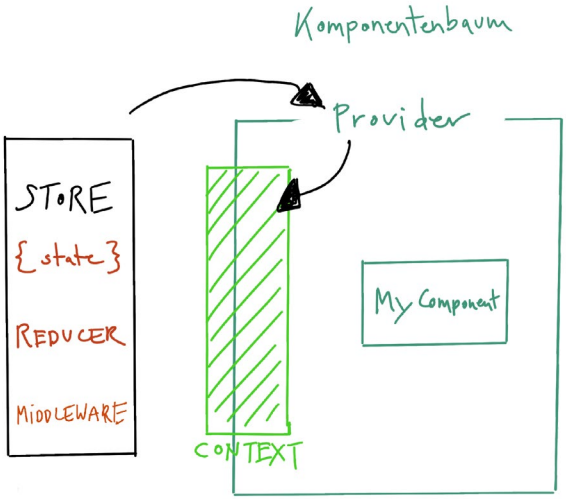
Mit React Redux wird die Komponente *Provider* eingebunden, die den Redux-Store für den Rest der Applikation verfügbar macht. Die Provider-Komponente wird auf der obersten Ebene des Komponentenbaums integriert:

```
1  import React from 'react';
2  import ReactDOM from 'react-dom';
3
4  import { Provider } from 'react-redux';
5  import store from 'App/store';
6
7  import App from 'App/components/App';
8
9  const rootElement = document.getElementById('root');
10
11 ReactDOM.render(
12   <Provider store={store}>
13     <App />
14   </Provider>,
15   rootElement
16 );
```

Zum Code →

PROVIDER

Über diese Provider-Komponente haben nun alle Unterkomponenten, die in der Provider-Komponente eingebunden sind, ***automagically*** über die Context-Schnittstelle von React Zugriff auf den Redux-Store.



Die Store-Methoden an React-Komponenten weiterreichen

Mit React-Redux müssen die eigenen React-Komponenten selbst niemals direkt auf den Store zugreifen. Dies erledigt ***connect*** von React-Redux für uns.

Connect ist eine Higher-Order-Komponente, eine Funktion, die eine React-Komponente mitgegeben bekommt und eine neue React-Komponente zurückgibt. Mit ***connect*** wird eine neue, mit dem Redux-Store verbundene Komponente erstellt, die auf Redux-Store Methoden wie ***dispatch*** und ***getState*** Zugriff hat.

```
const MyConnectedComponent = connect()(MyComponent);
```

Die übergebene React-Komponente wird nicht von ***connect*** geändert, sondern mit relevanten Props erweitert. Dadurch hat die mitgegebene React-Komponente die Möglichkeit, die für sie relevanten Daten aus dem State zu lesen und mittels Actions den State zu verändern.

Connect akzeptiert vier verschiedene Parameter, die alle optional sind:

```
const MyConnectedComponent = connect(mapStateToProps,
mapDispatchToProps, mergeProps, options)(MyComponent);
```

Für diesen Artikel schauen wir uns die ersten beiden Parameter an, da diese für die Anbindung an den Redux-Store relevant sind. Weitere Informationen zu den anderen Parametern gibt es auf der offiziellen Dokumentations-Seite von React-Redux.

mapStateToProps

Der *mapStateToProps*-Parameter ist eine Funktion, die jedes Mal aufgerufen wird, sobald sich der State geändert hat. Wurde *connect* mit *mapStateToProps* aufgerufen, verbindet sich die mit connect erstellte Komponente mit dem Redux-Store anhand der Methode *store.subscribe()*. Ändert sich der State, wird *mapStateToProps* mit dem State aufgerufen (*store.getState()*). Das Ergebnis von *mapStateToProps* ist ein Objekt, das als erweiterte Props an die gewrappte Komponente mitgegeben wird. Nur wenn sich die durch *mapStateToProps* zurückgegebenen State-Werte verändert haben, wird die verbundene Komponente mit den aktualisierten Props neu gerendert.

MapStateToProps akzeptiert zwei Parameter, wobei die zweite optional ist:

```
mapStateToProps: (state, ownProps) => object
```

Wird die *mapStateToProps*-Funktion mit nur einem Parameter deklariert, wird sie immer dann aufgerufen, wenn sich der Redux-State ändert. Das erste Argument ist in diesem Falle der gesamte Redux-State, bzw. das Ergebnis von *store.getState()*:

```
const mapStateToProps = state => ({ user: state.user });
```

```
import React from 'react';
import { Provider, connect } from 'react-redux';

const MyComponent = ({ user }) => .....

// in mapStateToProps definieren, welche state values für die verbundene Komponente relevant sind
const mapStateToProps = state => ({
  user: state.user
})

// mit connect eine verbundene Komponente erstellen
// die MyComponent neu rendert, sobald sich state.user ändert
const ConnectedComponent = connect(mapStateToProps)(MyComponent);

ReactDOM.render(
  <Provider store={store}>
    <ConnectedComponent />
  </Provider>,
  document.getElementById('root')
);
```

Zum Code →

Wird die *mapStateToProps*-Funktion mit zwei Parametern deklariert, wird sie weiterhin immer aufgerufen, wenn sich der Redux-State ändert oder wenn sich die Props der Elternkomponente ändern, die hier als *ownProps* an die verbundene Komponente von *connect* weitergegeben werden.

```
import React from 'react';
import { Provider, connect } from 'react-redux';

const MyComponent = ({ user }) => .....

// mapStateToProps mit zwei Parameter definieren
const mapStateToProps = (state, ownProps) => ({
  user: state.users[ownProps.id]
})

// mit dem Store verbundene Komponente erstellen
const ConnectedComponent = connect(mapStateToProps)(MyComponent);

ReactDOM.render(
  <Provider store={store}>
    <ConnectedComponent id="some-ID" /> // some-ID wird als ownProps an mapStateToProps runtergereicht
  </Provider>,
  document.getElementById('root')
);
```

Zum Code →

mapDispatchToProps

Als zweites Argument von *connect* wird *mapDispatchToProps* deklariert, was relevant für das Dispatchen von Actions an den Store ist.

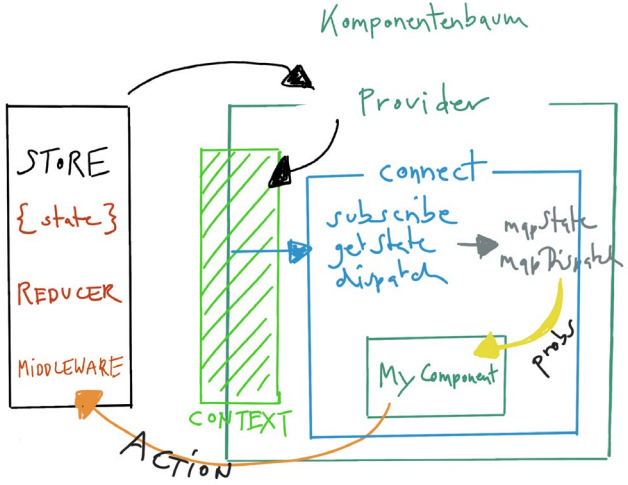
Wird *mapDispatchToProps* als Objekt mitgegeben, wird jedes Feld im Objekt als Action Creator behandelt und in der verbundenen Komponente von *connect* standardmäßig mit *store.dispatch* aufgerufen. Das bedeutet, dass man die bereits definierten Action Creators in *mapDispatchToProps* angeben kann, und *connect* kümmert sich darum, dass die verbundene Komponente diese Action bereits mit *dispatch* als Prop zur Verfügung gestellt bekommt.

```
import React from 'react';
import { Provider, connect } from 'react-redux';
import { setFirstName, setLastName, setAge } from 'App/store/user/actions';

// MyComponent, deren action props bereits mit dispatch gewrapped sind
const MyComponent = ({ setFirstName, setLastName, setAge }) => .....

// mapDispatchToProps als Objekt mit Action Creators
const mapDispatchToProps = {
  setFirstName,
  setLastName,
  setAge
})

// mit dem Store verbundene Komponente erstellen
const ConnectedComponent = connect(
  null, // in diesem Beispiel werden keine State Props mitgegeben
  mapDispatchToProps
)(MyComponent);


ReactDOM.render(
  <Provider store={store}>
    <ConnectedComponent />
  </Provider>,
  document.getElementById('root')
);
```

Zum Code →

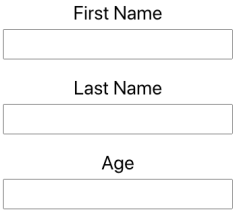
Hinweis: Gibt man keine Action Creators an *mapDispatchToProps* weiter, sondern simple Action-Objekte, oder möchte man auch in *mapDispatchToProps* auf *ownProps* (die übergebenen Props der Elternkomponente) zugreifen, muss man *mapStateToProps* als Funktion deklarieren und nicht als Objekt. Die offizielle Dokumentation empfiehlt allerdings die Objekt-Schreibweise mit Action Creators. Sollte dies aus den genannten Gründen nicht ausreichen, gibt es auf der offiziellen React-Redux Website weitere Informationen.

Wir haben gesehen, wie man mit React-Redux einzelne React-Komponenten mit dem Redux-Store verbinden kann. Durch die Provider-Komponente wird der Redux-Store über den React-Context an untere Komponenten weitergereicht. Mit der Funktion ***connect*** wird auf diesen Store im React-Context zugegriffen und, wenn ***mapStateToProps*** und/oder ***mapDispatchToProps*** definiert wurden, als Props an die gewrappte React-Komponente weitergegeben. Dadurch muss die eigentliche React-Komponente nicht direkt mit dem Redux-Store kommunizieren, da sich ***connect*** „under the hood" um die Verbindung kümmert.



Beispiel User Komponente

Um den bereits oben definierten User-State mit ***firstName***, ***lastName*** und ***age*** und die dazugehörigen Action Creators an React-Komponenten weiterzureichen, werden wir nun einige Komponenten definieren, die eine einfache Eingabeform rendern. Diese Eingabeform liest Daten aus dem Redux-State und verändert Daten im Redux-State bei jeder Texteingabe.



Als Erstes schreiben wir ein einfaches Eingabefeld als React-Komponente, die bei jeder Feldeingabe den Wert aufruft. Diese Komponente ist nicht mit dem Redux-Store verbunden:

```
1  // App/components/user/InputField.js
2
3  import React from 'react';
4
5  import styles from './User.module.css';
6
7
8  const InputField = ({ label, value, onChange }) => {
9    return (
10     <div className={styles.row}>
11       <label>{label}</label>
12       <input
13         className={styles.textbox}
14         aria-label={label}
15         value={value}
16         onChange={e => onChange(e.target.value)}
17       />
18     </div>
19   );
20 };
21
22 export default InputField;
```

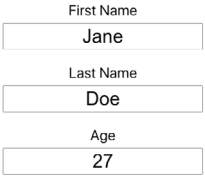
Zum Code →

Dieses Eingabefeld können wir nun für die drei User-State-Werte verwenden: *firstName*, *lastName* und *age*. Dafür schreiben wir eine User-Komponente, die dieses Eingabefeld drei Mal rendert und die mit dem Redux-Store durch die Funktion *connect* mit *mapStateToProps* und *mapDispatchToProps* verbunden ist:

```
// App/components/user/User.js

import React from 'react';
import { connect } from 'react-redux';

import { setFirstName, setLastName, setAge } from 'App/store/user/actions';
import InputField from './InputField';

const User = ({ user, setFirstName, setLastName, setAge }) => {
  return (
    <>
      <InputField label="First Name" value={user.first_name} onChange={setFirstName} />
      <InputField label="Last Name" value={user.last_name} onChange={setLastName} />
      <InputField label="Age" value={user.age} onChange={value => setAge(Number(value) || '')} />
    </>
  );
};

const mapStateToProps = (state) => {
  return {
    user: state.user
  }
}

const mapDispatchToProps = { setFirstName, setLastName, setAge }

export default connect(
  mapStateToProps,
  mapDispatchToProps
)(User)
```

Zum Code →

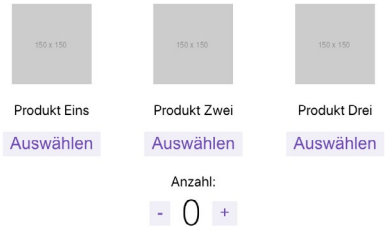
Die Werte der Eingabefelder werden aus dem Redux-State state.user gelesen und bei jeder Texteingabe wiederum durch die Actions geändert, was ein erneutes Rendering der Komponenten mit dem aktualisierten State-Wert zur Folge hat.



Beispiel Basket Komponente

Ähnlich gehen wir vor, wenn wir Komponenten mit dem basket-State verbinden wollen.

Als Erstes rendern wir eine Liste von Beispiel-Produkten, aus denen der User eines auswählen kann.



Dafür schreiben wir eine Komponente, die sich um das Rendering eines einzelnen Produktes kümmert.

Diese Komponente bekommt von ihrer Elternkomponente die Prop “name”, die in ***mapStateToProps*** als ***ownProp*** weitergereicht wird. Darüber hinaus wird in ***mapStateToProps*** überprüft, ob diese Komponente gerade im State ausgewählt wurde, indem der Name über die ***ownProps*** mit dem aktuellen Wert im ***state.basket*** verglichen wird.

Die relevante Action, um das Produkt auszuwählen, kann in ***mapDispatch-ToProps*** weitergereicht werden. Sobald ein Produkt ausgewählt wird, wird der entsprechende Produktname in den Redux-State ***state.basket.product-Name*** gesetzt und die Komponente rendert sich entsprechend neu, da sie nun ausgewählt wurde (***isSelected === true***).

```
// App/components/basket/Product.js

import React from 'react';
import { connect } from 'react-redux';

import { setProductName } from './actions';

import styles from './Basket.module.css';

const Product = ({ name, isSelected, setProductName }) => {
  return (
    <div className={isSelected ? styles.selected : ''}>
      <img src="https://via.placeholder.com/150" alt={name} />
      <p>{name}</p>
      <button
          className={styles.button}
          aria-label="Produkt auswählen"
          onClick={() => setProductName(name)}
        >
          Auswählen
      </button>
    </div>
  );
};

const mapStateToProps = (state, ownProps) => {
  return {
    name: ownProps.name,
    isSelected: state.basket.productName === ownProps.name
  }
}

const mapDispatchToProps = { setProductName }

export default connect(
  mapStateToProps,
  mapDispatchToProps
)(Product)
```

Zum Code →

Um dem User anzuzeigen, welcher Produktname gerade ausgewählt ist, schreiben wir noch eine weitere Komponente, die sich nur um diese Anzeige kümmert, indem sie den Namen aus *state.basket* liest:

```
// App/components/basket/SelectedProduct.js

import React from 'react';
import { connect } from 'react-redux';

const SelectedProduct = ({ productName }) => {
  if (!productName) return null;

  return <p><strong>Dein gewähltes Produkt:</strong> {productName}</p>;
};

const mapStateToProps = state => {
  return {
    productName: state.basket.productName
  }
};

export default connect(mapStateToProps)(SelectedProduct);
```

Zum Code →

Auch wenn es sich bei der Komponente *"SelectedProduct"* um eine sehr kleine Komponente handelt, ist es grundsätzlich in Ordnung, verschiedene Komponenten mit *connect* aufzurufen bzw. verschachtelte Komponenten erneut direkt mit dem Store zu verbinden. Es handelt sich hierbei nicht um Performance-Einbußen, wie vor einiger Zeit teilweise noch angenommen wurde. Auf der offiziellen Dokumentations-Seite von Redux wird dies detaillierter beschrieben.

Mit dieser Architektur muss die Elternkomponente des Baskets nicht mit dem Redux-Store verbunden werden, da die Kinder-Komponenten bereits “schlau genug” sind und selber auf den State schauen oder diesen verändern. Allerdings erwarten die einzelnen Produktkomponenten eine “name” Prop von der Elternkomponente:

```
// App/components/basket/Products.js

import React from 'react';

import Product from './Product';
import SelectedProduct from './SelectedProduct';

import styles from './Basket.module.css';

const Products = () => {
  return (
    <>
      <div className={styles.products}>
        <Product name="Produkt Eins" />
        <Product name="Produkt Zwei" />
        <Product name="Produkt Drei" />
      </div>
      <SelectedProduct />
    </>
  );
};

export default Products;
```

Zum Code →

Für die Auswahl der Produkt-Anzahl erstellen wir eine einfache Komponente, die ebenso mit dem Redux-Store connected ist und den aktuellen *amount* aus dem basket-State liest und mit den beiden Actions *incrementAmount* und *decrementAmount* diesen Wert aktualisieren kann:

```jsx
import React from 'react';
import { connect } from 'react-redux';

import { incrementAmount, decrementAmount } from './actions';
import styles from './Basket.module.css';

const Amount = ({ amount, incrementAmount, decrementAmount }) => {

  return (
    <>
      <span>Anzahl:</span>
      <div className={styles.row}>
        <button
          className={styles.button}
          aria-label="Decrement value"
          onClick={() => decrementAmount()}
        >
          -
        </button>
        <span className={styles.value}>{amount}</span>
        <button
          className={styles.button}
          aria-label="Increment value"
          onClick={() => incrementAmount()}
        >
          +
        </button>
      </div>
    </>
  );
}

const mapStateToProps = (state) => {
  return {
    amount: state.basket.amount
  }
}

const mapDispatchToProps = { incrementAmount, decrementAmount }

export default connect(
  mapStateToProps,
  mapDispatchToProps
)(Amount)
```

Zum Code →

React-Redux mit Hooks

Seit der Version 7.1.0 von React-Redux gibt es eine Hook-Schnittstelle, die als Alternative zu der bestehenden *connect*-Funktion genutzt werden kann. Mit diesen neuen Hooks ist es nun möglich, eine React-Komponente direkt mit dem Redux-Store zu verbinden, ohne diese vorher mit *connect* aufzurufen. Dies macht auch das Definieren von *mapStateToProps* und *mapDispatchToProps* obsolet und reduziert einiges an Code.

useSelector

Mit dem *useSelector*-Hook ist es möglich, Daten aus dem Redux-State zu lesen. Somit verhält sich der Selector ähnlich wie *mapStateToProps* von der *connect*-Funktion. Ein Selector-Hook wird mit dem gesamten Redux-

State aufgerufen und das jedes Mal, wenn sich die Komponente neu rendert und wenn sich der Redux-State durch das dispatchen einer Action ändert. Dabei vergleicht der Selector, ob sich der referenzierte State-Wert geändert hat und forciert nur dann eine erneutes Rendering. Da der Selector direkt in der React-Komponente deklariert wird und dadurch mehrfach aufgerufen werden kann, muss darauf geachtet werden, dass die Selector-Funktion eine pure *function* ist. Das heißt, die Funktion sollte keinen internen State halten oder Seiteneffekte aufrufen und immer denselben Wert zurückgeben (den State-Value, der gelesen wird).

Zum Vergleich hier eine einfache Komponente mit *connect* und *mapStateToProps*:

```
1  import React from 'react';
2  import { connect } from 'react-redux';
3
4  const Products = ({ productName }) => {
5    return (
6      <p><strong>Dein gewähltes Produkt:</strong> {productName}</p>
7    );
8  };
9
10 const mapStateToProps = (state) => {
11   return {
12     productName: state.basket.productName
13   }
14 }
15
16 export default connect(mapStateToProps)(Products);
```

Zum Code →

und mit Hooks:

```
1  import React from 'react';
2  import { useSelector } from 'react-redux';
3
4  const Products = () => {
5    const productName = useSelector(state => state.basket.productName);
6
7    return (
8      <p><strong>Dein gewähltes Produkt:</strong> {productName}</p>
9    );
10 };
11
12 export default Products;
```

Zum Code →

useDispatch

Der *useDispatch*-Hook gibt eine Referenz zur *store.dispatch()*-Methode zurück und kann somit direkt in einer React-Komponente eingebunden werden. Im Gegensatz zum *useSelector*-Hook wird die Referenz zu *dispatch* nur einmal erstellt, und man muss sich keine Gedanken um ein erneutes Rendering machen.

Zum Vergleich hier eine einfache Komponente mit *connect* und *mapDispatchToProps*:

```
import React from 'react';
import { connect } from 'react-redux';

import { selectProduct } from './actions';

const Product = ({ selectProduct }) => {
  return (
    <button onClick={() => selectProduct()}>Product Select</button>
  );
};

const mapDispatchToProps = {
  selectProduct
}

export default connect(null, mapDispatchToProps)(Products);
```

Zum Code →

und mit Hooks:

```
import React from 'react';
import { useDispatch } from 'react-redux';

import { selectProduct } from './actions';

const Product = () => {
  const dispatch = useDispatch();

  return (
    <button onClick={() => dispatch(selectProduct())}>Product Select</button>
  );
};

export default Products;
```

Zum Code →

Beispiel Basket Komponente mit Hooks

Zur Veranschaulichung bauen wir zwei Komponenten mit den Hooks von React-Redux um.

Die Komponente “Produkt” mit *connect*:

```
// App/components/basket/Product.js

import React from 'react';
import { connect } from 'react-redux';

import { setProductName } from './actions';

import styles from './Basket.module.css';

const Product = ({ name, isSelected, setProductName }) => {
  return (
    <div className={isSelected ? styles.selected : ''}>
      <img src="https://via.placeholder.com/150" alt={name} />
      <p>{name}</p>
      <button
          className={styles.button}
          aria-label="Produkt auswählen"
          onClick={() => setProductName(name)}
        >
          Auswählen
      </button>
    </div>
  );
};

const mapStateToProps = (state, ownProps) => {
  return {
    name: ownProps.name,
    isSelected: state.basket.productName === ownProps.name
  }
}

const mapDispatchToProps = { setProductName }

export default connect(
  mapStateToProps,
  mapDispatchToProps
)(Product)
```

Zum Code →

und umgebaut mit *useSelector* und *useDispatch*:

```
// App/components/basket/Products.js

import React from 'react';
import { useSelector, useDispatch } from 'react-redux';

import { setProductName } from './actions';

import styles from './Basket.module.css';

// prop "name" wird von der Elternkomponente weitergereicht
const Product = ({ name }) => {

  const dispatch = useDispatch(); // Referenz auf store.dispatch erstellen

  // mit einer Selector Funktion Daten aus dem Redux-State lesen
  const selectedProductNameInState = useSelector(state => state.basket.productName);
  const isSelected = name === selectedProductNameInState;

  return (
    <div className={isSelected ? styles.selected : ''}>
      <img src="https://via.placeholder.com/150" alt={name} />
      <p>{name}</p>
      <button
          className={styles.button}
          aria-label="Add to basket"
          onClick={() => dispatch(setProductName(name))}
        >
          Auswählen
      </button>
    </div>
  );
};

export default Product;
```

Zum Code →

Die Komponente “SelectedProdukt” mit *connect*:

```
// App/components/basket/SelectedProduct.js

import React from 'react';
import { connect } from 'react-redux';

const SelectedProduct = ({ productName }) => {
  if (!productName) return null;

  return <p><strong>Dein gewähltes Produkt:</strong> {productName}</p>;
};

const mapStateToProps = state => {
  return {
    productName: state.basket.productName
  }
};

export default connect(mapStateToProps)(SelectedProduct);
```

Zum Code →

und umgebaut mit *useSelector*:

```
// App/components/basket/SelectedProduct.js

import React from 'react';
import { useSelector } from 'react-redux';

const SelectedProduct = () => {
  const productName = useSelector(state => state.basket.productName);

  if (!productName) return null;

  return <p><strong>Dein gewähltes Produkt:</strong> {productName}</p>;
};

export default SelectedProduct;
```

Zum Code →

Die Elternkomponente kann so bleiben wie vorher, da sie auch nach Umbau nicht an den Redux-Store angebunden ist:

```
// App/components/basket/Products.js

import React from 'react';

import Product from './Product';
import SelectedProduct from './SelectedProduct';

import styles from './Basket.module.css';

const Products = () => {
  return (
    <>
      <div className={styles.products}>
        <Product name="Produkt Eins" />
        <Product name="Produkt Zwei" />
        <Product name="Produkt Drei" />
      </div>
      <SelectedProduct />
    </>
  );
};

export default Products;
```

Zum Code →

Asynchrone Action mit Redux-Thunk

Da wir bereits bei der Erstellung des Stores die Middleware Redux-Thunk eingebunden haben, ist es möglich, asynchrone Actions bzw. Thunk Actions an den Store zu schicken.

Zur Veranschaulichung erstellen wir hierfür eine weitere Komponente, die einen Submit-Button rendert und beim *onClick*-Handler eine asynchrone Action dispatched:

```
import React from 'react';
import { useDispatch } from 'react-redux';

import { submit } from './actions';

import styles from './Submit.module.css';

const Submit = () => {
  const dispatch = useDispatch();

  return (
      <button
        className={styles.button}
        onClick={() => dispatch(submit())}>
        Absenden
      </button>
  );
};

export default Submit;
```

Zum Code →

Wie bereits oben beschrieben, sorgt die Middleware Redux-Thunk dafür, wenn *dispatch* mit einem Action Creator aufgerufen wird, der nicht ein Action-Objekt, sondern eine Funktion zurückgibt, dass dann diese Funktion als Thunk-Action erkannt und mit *dispatch* und *getState* aufgerufen wird. Dadurch ist es möglich, innerhalb dieser Thunk-Action auf den aktuellen State zuzugreifen und andere Actions mit *dispatch* aufzurufen.

In unserem Beispiel erstellen wir die Thunk-Action “submit”, die aus dem Redux-State die User-Daten und die Basket-Daten liest und diese an einen Server mit *fetch* schickt.

```
// App/store/submit/actions

// asynchrone Thunk Action

export const submit = () => {
  return async (dispatch, getState) => {
    const state = getState();
    const user = state.user;
    const basket = state.basket;

    try {
      const response = await fetch('https://your-url.com', {
        method: 'POST',
        body: JSON.stringify({
          user,
          basket
        }),
        headers: {
          'Content-type': 'application/json;'
        }
      });

      if (response.ok) {
        dispatch({
          type: 'POST_SUCCESS'
        });
      } else {
        dispatch({
          type: 'POST_FAILURE'
        });
      }
    } catch (error) {
      dispatch({
        type: 'REQUEST_ERROR'
      });
    }
  };
};
```

Zum Code →

Fazit

Mit der Einbindung von Redux in eine React-Applikation bekommt man ein mächtiges Tool geliefert, um den Applikations-State zu organisieren. Auch wenn es einiges an Konzepten gibt, die man eingangs lernen muss und somit der Zugang gerade für Anfänger nicht einfach ist, lohnt es sich auf lange Sicht, mit dem globalen State-Management von Redux zu arbeiten, um eine Applikation übersichtlicher zu gestalten.

Das direkte Verbinden einer React-Komponente mit dem globalen Redux-State verhindert das Durchreichen verschiedener Props durch mehrere Komponentenebenen und macht dadurch einzelne Komponenten unabhängig von ihrer Verschachtelung im Komponentenbaum.

Globales State-Management mit Redux erfordert einiges an Boilerplate, um den Redux-Store mit Actions und Reducers zu implementieren. Doch hat man einmal das Redux-Konzept verinnerlicht, geht auch das leicht von der Hand. Mit der Installation von Redux DevTools sollte es gerade Anfängern leichter erscheinen, zu verstehen, wie der Redux-Flow mit Actions, Reducers und State funktioniert. Zudem wurde während der Erstellung dieses Kapitels die offizielle Redux-Dokumentation mit dem Fokus auf Redux-Toolkit aktualisiert, was ebenso die Hürde für Anfänger verringern soll.

Der Autor

Jan Bussieck

Jan Bussieck ist selbstständiger Webentwickler, Berater und React-Trainer. Wenn er sich nicht gerade die Zähne daran ausbeißt, moderne Webtechnologien in die Enterprise zu bringen, entwickelt er neue Materialien für React-Workshops oder verbreitet tendenziöse Schriften über die Webentwicklung auf seinem Blog.

React Fiber: Was verbirgt sich hinter dem neuen Kernalgorithmus?

Einer der Vorteile einer deklarativen UI, der wenig besprochen wird, ist, dass wir Entwickler nicht mehr entscheiden müssen, ***wann*** wir Updates am DOM vornehmen. Wir übergeben React einfach ein paar Funktionen (oder Klassen) in der Form von Komponenten, die beschreiben, ***was*** gerendert werden soll, und es ist Reacts Aufgabe, diese aufzurufen und die resultierenden Updates auszuführen.

Wie wir im Folgenden sehen werden, eröffnet diese subtile Unterscheidung ungeahnte Möglichkeiten. Da es nun in Reacts Zuständigkeitsbereich liegt, Komponentenfunktion aufzurufen, kann React dies auch zu einem späteren Zeitpunkt tun oder die Ausführung einiger Funktionen priorisieren.

Dies ist eine zentrale Idee der React Philosophie. In Push-Ansätzen werden Updates angestoßen, sobald neue Daten verfügbar sind. Das UI-Framework mag den Datenfluss vielleicht kontrollieren, dass und wann etwas passiert, wird von der Datenseite bestimmt. React hingegen verfolgt einen Pull-Ansatz und bestimmt eigenmächtig, was wann passiert. So kann React entscheiden, dass es wichtiger ist, Updates eines kontrollierten Text-Inputs anzuzeigen als die Daten, die soeben vom Server gekommen sind.

Bis React Version 16 wurde diese Möglichkeit jedoch nicht ausgenutzt. Der komplette Renderzyklus wurde innerhalb eines Funktionsaufrufs im Hauptthread vollzogen, wodurch dieser für die Dauer des Renderns blockiert war und keine anderen Aufgaben abarbeiten konnte, weder neue

Updates von React noch andere Funktionsaufrufe. Nach einem Update des States einer Komponente (z. B. durch *setState*) wird ein Re-render angestoßen. Dieser Prozess basiert auf Reacts Kernalgorithmus, welcher unter Zuhilfenahme einiger Heuristiken aus der Graphentheorie jeden Knoten unserer Anwendung durchläuft, um einen Diff aus unserem alten und neuen Zustandsbaum zu erstellen. Dieser Diff enthält die minimale (zumindest approximativ) Menge von Änderungen, die React am DOM vornehmen muss, um vom alten in den neuen Zustand überzugehen. Dieser Prozess heißt bei React Reconciliation, also Versöhnung des alten States mit dem neuen.

React Fiber ist ein kompletter Rewrite dieses Reconciliation-Algorithmus, dessen Ziel es ist, die Vorteile des oben beschrieben Pull-Ansatzes auszunutzen.

Ein minimales Beispiel

Um das Ganze konkreter zu machen, schauen wir uns im Folgenden ein einfaches Beispiel, den Klick Zähler an (wenn man so will, ist das der Drosophila Melanogaster der React Tutorials):

```
1  import React, { useState } from 'react';
2
3  const ClickCounter = () => {
4    const [clickCount, setClickCount] = useState(0);
5    const handleClick = () => setClickCount(prevCount => prevCount + 1);
6    return [
7      <button key='btn' onClick={handleClick}>Zähler hochzählen</button>,
8      <span key='count'>{clickCount} mal geklickt</span>
9    ]
10 };
```

Zum Code →

Wir definieren eine funktionale Komponente, welche einen Button und ein *span*-Element zurückgibt. Beim Klick auf den Button wird über einen Event Handler der Komponenten State geupdatet und als Resultat wiederum der Text des *span*-Elementes.

Reacts Reconciliation-Algorithmus muss also beim initialen Rendern und anschließenden Update unserer kleinen Anwendung:

- den *clickCount* Wert des Komponenten State updaten
- die children des *ClickCounter* und deren Props abrufen und vergleichen
- die Props des *span*-Elementes updaten

Es finden noch weitere Buchhaltungsaktivitäten während der Reconciliation statt. So muss zum Beispiel auch der *useState hook* angelegt und in einer Queue getrackt werden.

Alle Aktivitäten des Fiber-Algorithmus werden in der Fiber-Architektur als “work” bezeichnet, welches sich wiederum in “units of work” runterbrechen lässt.

Doch bevor wir im Detail verstehen können, wie der Fiber-Algorithmus sich diese “units of work” einteilt und was genau mit ihnen geschieht, müssen wir noch eine kurze Exkursion machen und uns die Datenstrukturen, die React intern benutzt, genauer ansehen.

Die Datenstrukturen hinter React-Komponenten

Dieser Datenstruktur kommen wir einen Schritt näher, wenn wir schauen, womit der JSX Compiler die Komponenten, die unsere *ClickCounter* Komponente zurückgibt, ersetzt. Wir rufen uns in Erinnerung, dass JSX Elemente wie *<button></button>* nur syntaktischer Zucker für den Aufruf von *React.createElement* sind:

```
...
return [
  React.createElement(
    "button",
    {
      key: "btn",
      onClick: handleClick,
    },
    "Zähler hochzählen"
  ),
  React.createElement(
    "span",
    {
      key: "count",
    },
    `${clickCount} mal geklickt`
  ),
];
...
```

Zum Code →

Die Aufrufe von *React.createElement* wiederum erstellen eine solche Datenstruktur:

```
[
  {
    $$typeof: Symbol(react.element),
    type: "button",
    key: "btn",
    props: {
      children: "Zähler hochzählen",
      onClick: () => { ... },
    },
  },
  {
    $$typeof: Symbol(react.element),
    type: "span",
    key: "count",
    props: {
      children: 0,
    },
  },
];
```

Zum Code →

React fügt den Key *$$typeof* hinzu, um die Objekte als React Elemente zu identifizieren. Die *type*, *key* und *prop* Eigenschaften beschrieben das jeweilige Element und entsprechen den an das *React.createElement* übergebenen Werten. Das React Element für das *ClickCounter* Element enthält weder Props noch Keys:

```
{
  $$typeof: Symbol(react.element),
  key: null,
  props: {},
  ref: null,
  type: ClickCounter
}
```

Zum Code →

Während der Reconciliation wird jede Komponente des Komponentenbaumes aufgerufen und aus den Objektrepräsentationen der zurückgegebenen Elemente wiederum Fiber-Knoten erstellt und in einer Baumstruktur zusammengeführt. Jedes React Element erhält so einen korrespondieren Fiber-Knoten, welcher anders als die Elemente nicht bei jedem Render neu erstellt werden.

Es sind genau diese Fiber-Knoten, die eine 'unit of work' verkapseln. Beim *ClickCounter* bestünden diese daraus, die Funktion aufzurufen und einen *useState*-Hook zu allozieren, wobei beim *span* DOM Veränderungen vorgenommen werden müssen.

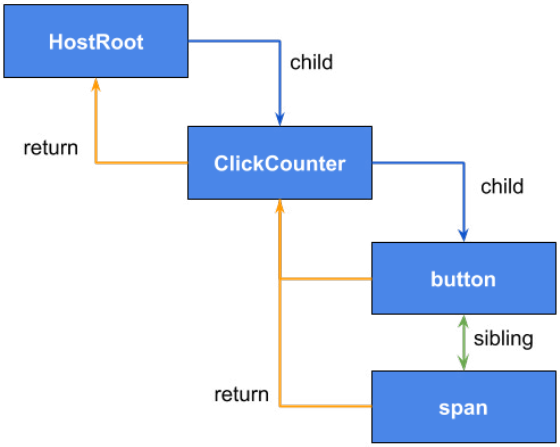
Hierin liegt der ganz Trick der Fiber-Architektur: Da wir die Render-Arbeit auf in sich geschlossene units of work herunterbrechen können, können wir diese einfach tracken, schedulen, pausieren oder abbrechen.

Wenn beim initialen Render ein React-Element in einen Fiber-Knoten überführt wird, werden dessen Daten kopiert, der Fiber-Knoten von React anschließend wiederverwendet und nur notwendige Updates

vorgenommen, um ihn mit dem korrespondierenden Element synchron zu halten. Zudem können Fiber-Knoten in einer Liste neu geordnet (hierin liegt die Bedeutung des *key prop*) oder gelöscht werden, falls das entsprechende React-Element bei einem Render nicht mehr zurückgegeben wird.

[Eine Übersicht aller Aktivitäten eines Fiber-Knoten findet sich in der *ChildReconciler* Funktion.]

Der Fiber-Baum unserer *ClickCounter*-Anwendung kann wie folgt dargestellt werden:



Alle Fiber-Knoten sind als verkettete Liste dargestellt mit Referenzen auf die jeweiligen *child*, *sibling* oder *return* Knoten.

Der Work-in-Progress Baum

Nach dem initialen Render erstellt React einen Fiber-Baum, der den aktuellen Zustand unserer Anwendung reflektiert. Dieser Baum wird als ***current*** bezeichnet. Sobald dieser Zustand sich verändert, wird ein ***workInProgress*** Baum angelegt, welcher den neuen, jedoch noch nicht im DOM dargestellten Zustand repräsentiert.

Die in einem Render-Zyklus zu erledigende Arbeit wird nur am ***workInProgress*** Baum vorgenommen. React iteriert über den ***current***-Baum und erstellt für jeden Knoten einen Klon im ***workInProgress*** Baum. Die Daten dieses neuen Knotens erhält React wiederum durch das Rendern der React-Komponente. Erst wenn alle Komponenten gerendert, und die entsprechenden Updates am ***workInProgress*** Baum vorgenommen wurden, wird dieser in das DOM gerendert und der Baum wird zum neuen ***current***.

Ein weiteres Kernprinzip von React ist ***Konsistenz***. Daher wird das Rendern des DOMs als atomare Operation verstanden, welche keine partiellen Updates zulässt. So dient der ***workInProgress*** Baum als vorläufiger Entwurf, welcher dem Nutzer unserer Anwendung verborgen bleibt, bis nicht wirklich alle Änderungen verarbeitet wurden. Darin spiegelt sich auch der oben genannte Pull-Ansatz wieder, da neue Daten von React nicht direkt in DOM Updates übersetzt werden, sondern React entscheidet, welche Bedingungen erfüllt sein müssen, bevor ein neuer Zustand bereit ist, um final gerendert zu werden.

Der Work-Loop

Wir klicken nun den Button, rufen *setClickCount* auf, und die Funktion, die wir übergeben haben, wird einer Update-Queue hinzugefügt, bevor React mit dem Scheduling der units of work beginnt.

Dabei benutzt React die *requestIdleCallback* API und fragt damit den JavaScript Haupt-Thread "lass mich wissen, wenn du etwas Zeit übrig hast und erledige dann folgende Aufgabe". Der Haupt-Thread meldet sich zurück mit einer Zeitangabe, die React zur Verfügung steht.

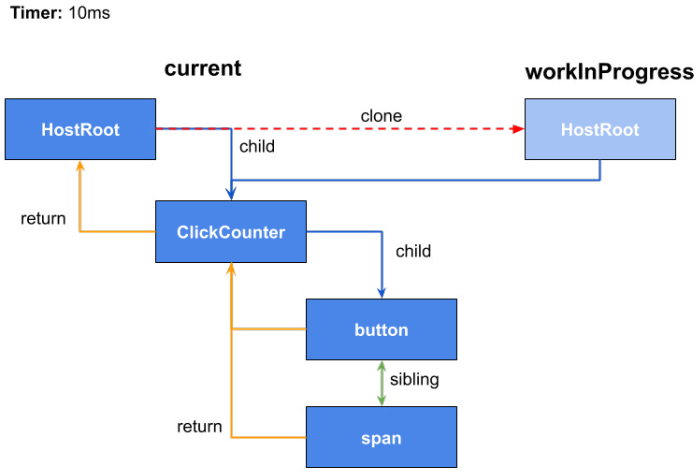
Um den *workInProgress* Baum aufzubauen und über die einzelnen Fiber-Knoten zu iterieren, benutzt React den Work-Loop. Der Kern dieser Funktion ist denkbar simpel:

```
1  function workLoop(isAsync) {
2    // ...
3    while (nextUnitOfWork !== null && !shouldYield()) {
4      nextUnitOfWork = performUnitOfWork(nextUnitOfWork);
5    }
6    // ...
7  }
```

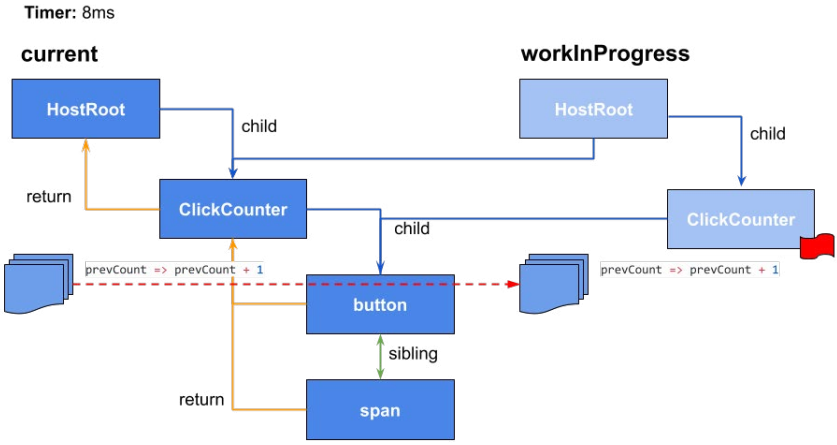
Zum Code →

React checkt, ob noch units of work abzuarbeiten sind und via *shouldYield()*, ob noch genügend vom Haupt-Thread zur Verfügung gestellte Zeit bleibt, und arbeitet die nächste unit of work ab, falls beide Bedingungen erfüllt sind.

Nehmen wir an, der Haupt-Thread stellt 10ms zur Verfügung. React beginnt damit den *workInProgress* Baum aufzubauen:

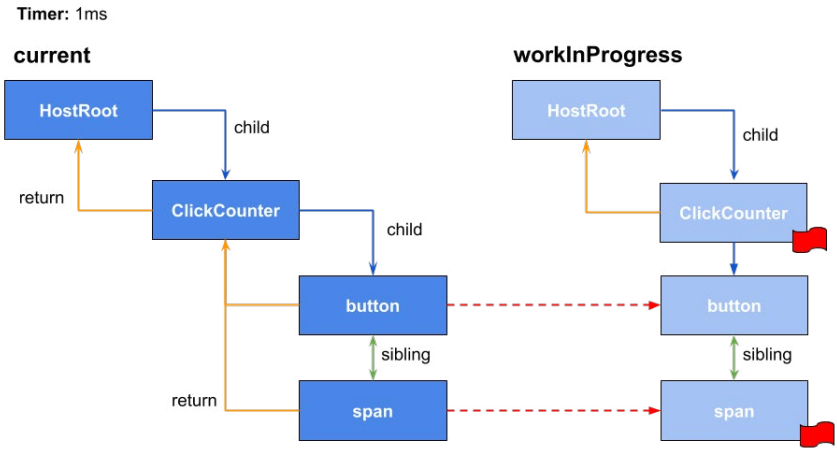


Zuerst wird der *HostRoot*-Knoten geklont. Der Pointer zu dem Child-Knoten im current Baum wird dabei mitkopiert. Der *ClickCounter*-Knoten wird als nächste "unit of work" zurückgeben und der Timer aktualisiert:



Der *ClickCounter* hat eine *UpdateQueue*, die ebenfalls geklont wird. React führt das Update aus, ändert den State des *ClickCounter*-Knotens und versieht ihn mit einem Effect-Tag, welches angibt, das DOM Änderungen vorgenommen werden müssen. Um den Baum weiter traversieren zu können, ruft React die *ClickCounter*-Funktion mit dem aktualisierten State auf und erhält ein Array von Kinder-Elementen.

Diese werden nun mit den Knoten des *current*-Baumes verglichen (da wir ein explizites key Prop gesetzt haben, wird dieses für den Vergleich verwendet), um zu entscheiden, ob die Kinder-Knoten übernommen werden können. In unserem Fall haben sich die *children* nicht verändern und werden geklont:



Die State-Änderungen des Eltern-Knoten führen dazu, dass die *children* des *span*-Knotens sich ebenfalls ändern und auch hier ein Effect-Tag hinzugefügt wird. Bei der nächsten Iteration des Work-Loops stellt React

fest, dass keine ausreichende Zeit mehr für die nächste unit of work verbleibt, ruft ***requestIdleCallback*** erneut mit einem Callback für die übrige Arbeit auf und gibt die Kontrolle wieder an den Haupt-Thread ab.

Dieser kann nun andere Aufgaben aus dem Haupt-Event-Loop abarbeiten. Ist inzwischen beispielsweise eine Animation oder Layout-Änderung eingegangen, können diese nun bearbeitet werden. React blockiert den Haupt-Thread nicht mehr, und die Anwendung bleibt auch nach einem angestoßenen Render-Zyklus responsive.

Ist der Haupt-Thread mit dieser Arbeit fertig, wird Reacts Callback aufgerufen, und das Rendern kann abgeschlossen werden. Dazu rufen die Kinderknoten auf ihren Eltern ***complete*** auf und fügen die in dem Effect-Tags vermerkten Änderungen zu denen ihrer Eltern hinzu.

Die finale Liste der Effects hängt nun an der ***HostRoot***, welche als ***pendingCommit*** markiert wird, und bietet die Grundlage für die im nächsten Schritt vorgenommenen Änderungen am DOM.

Diese werden, wie bereits beschrieben, als atomer Commit vorgenommen, damit keine UI-Inkonsistenzen auftreten können. Sobald das DOM dem ***workInProgress*** Baum entspricht, wird dieser zum aktuellen current Baum. Dazu wird lediglich der ***current*** Pointer umgebogen und die bereits vorhandenen Objekte wiederverwendet.

Es soll hier noch erwähnt sein, dass React, da es durch Fiber möglich ist, Rendering zu unterbrechen, unterschiedlichen Updates unterschiedliche Prioritäten zuweisen kann. Es soll geplant sein, diese künftig auch vom Anwender bestimmen zu lassen.

Anwendung: Error Boundaries

Der React Fiber Rewrite bringt eine weitere praktische Neuerung mit sich. Da einzelne Komponente nun als Fiber-Knoten separat aufgerufen werden und nicht mehr innerhalb einer Funktion, welche den Komponentenbaum traversiert, können auch Exceptions gesondert gefangen und behandelt werden.

Dazu stellt React ab Version 16 die Lifecycle-Methoden *getDerivedStateFromError()* und *componentDidCatch()* zur Verfügung, mit Hilfe derer sich eine Wrapper-Komponente definieren lässt, welche Exceptions, die zur Laufzeit in den Kinder-Komponenten auftreten, fangen und eine Fallback-UI definieren kann.

Bauen wir beispielsweise einen fiesen Bug in unsere *ClickCounter* Komponente ein, welcher dazu führt, dass nach dem dritten Klick eine Exception geworfen wird:

```
1  const ClickCounterWithError = () => {
2    const [clickCount, setClickCount] = useState(0);
3    const handleClick = () => setClickCount(prevCount => prevCount + 1);
4    if (clickCount > 3) {
5      throw new Error('crash me');
6    }
7    return [
8      <button key='btn' onClick={handleClick}>Zähler hochzählen</button>,
9      <span key='count'>{clickCount} mal geklickt</span>
10   ]
11 };
```

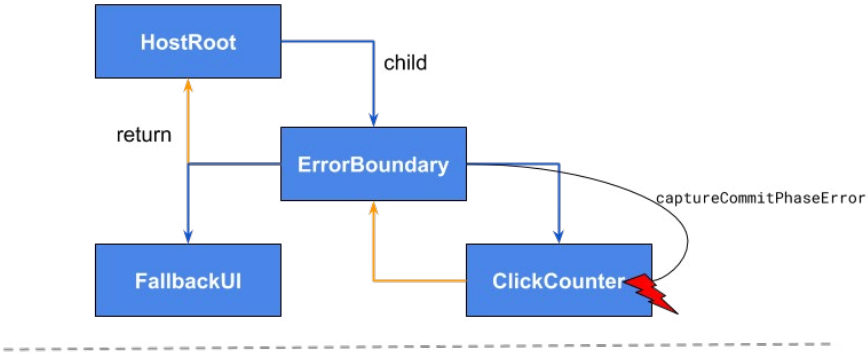
Zum Code →

Dann können wir mit einer ***ErrorBoundary*** Komponente verhindern, dass dadurch unsere gesamte Anwendung crasht:

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // Update state so the next render will show the fallback UI.
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
    // You can also log the error to an error reporting service
    console.log(error, errorInfo);
  }

  render() {
    console.log('has error', this.state.hasError)
    if (this.state.hasError) {
      // You can render any custom fallback UI
      return <span>I crashed</span>;
    }

    return this.props.children;
  }
}

const ClickCounterSave = () =>
  <ErrorBoundary>
    <ClickCounterWithError />
  </ErrorBoundary>
```

Zum Code →

In unserem Fiber-Baum sitzt nun ein weiterer Knoten zwischen der *HostRoot* und der *ClickCounter* Komponente. Wenn der Work-Loop beim Ausführen der Effektliste eine Exception fängt, wird die Funktion *captureCommitPhaseError* mit dem fehlerverursachenden Fiber-Knoten aufgerufen und so lange über dessen Elternknoten iteriert, bis eine Komponente gefunden wird, die die *ErrorBoundary* Lifecycle-Methoden definiert.



Fazit

Wir sehen, React Fiber treibt das React-Paradigma einen weiteren Schritt voran und liefert uns Optimierungen, die beim Bau komplexer UIs helfen.

Indem React den Renderprozess in kleinere unabhängige Einheiten aufteilt, ist es möglich, effizienter mit den vom Javascript Haupt-Thread zur Verfügung gestellten Ressourcen umzugehen und ermöglicht uns Entwicklern somit, performantere und responsive Anwendungen zu schreiben. Zudem machen Error-Boundaries unsere React-Anwendungen resilienter und nutzerfreundlicher. Hat man schon eine Zeit lang damit gearbeitet, scheint es ein Relikt aus dunkler Vorzeit, die App bei jedem Laufzeitfehler einfach crashen zu lassen.

Wir können gespannt sein, welche weiteren APIs uns in künftigen Versionen auf Grundlage von Fiber bereitgestellt werden. Je mächtiger und komplexer die Abstraktionen, mit denen wir arbeiten, jedoch werden, desto wichtiger wird es, dass wir verstehen, was sich dahinter verbirgt. Ich hoffe dieser Artikel hat einen ersten solchen Blick hinter den Vorhang geboten.

Host Europe

Sie haben weitere Fragen?

Unser Sales Team ist gerne für Sie da.

0800 626 4624

www.hosteurope.de