Kostenloses eBook

LERNEN React

Free unaffiliated eBook created from
Stack Overflow contributors.

#reactjs

Inhaltsverzeichnis

Über

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: react

It is an unofficial and free React ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official React.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Kapitel 1: Erste Schritte mit React

Bemerkungen

React ist eine deklarative, komponentenbasierte JavaScript-Bibliothek, die zum Erstellen von Benutzeroberflächen verwendet wird.

MVC - Framework wie Funktionalitäten zu erreichen in Reaktion, Entwickler verwenden es in Verbindung mit einem Flux Geschmack der Wahl, zB Redux .

Versionen

Ausführung	Veröffentlichungsdatum
0,3,0	2013-05-29
0,4,0	2013-07-17
0,5,0	2013-10-16
0,8,0	2013-12-19
0,9,0	2014-02-20
0,10,0	2014-03-21
0,11,0	2014-07-17
0,12,0	2014-10-28
0,13,0	2015-03-10
0,14,0	2015-10-07
15.0.0	2016-04-07
15.1.0	2016-05-20
15.2.0	2016-07-01
15.2.1	2016-07-08
15.3.0	2016-07-29
15.3.1	2016-08-19
15.3.2	2016-09-19

Ausführung	Veröffentlichungsdatum
15.4.0	2016-11-16
15.4.1	2016-11-23
15.4.2	2017-01-06
15.5.0	2017-04-07
15.6.0	2017-06-13

Examples

Installation oder Setup

ReactJS ist eine JavaScript-Bibliothek, die in einer einzelnen Datei `react-<version>.js` kann in jede HTML-Seite `react-<version>.js` werden. `react-dom-<version>.js` auch die React-DOM-Bibliothek `react-dom-<version>.js` zusammen mit der React `react-dom-<version>.js` installiert:

Grundlegende Inklusion

```
<!DOCTYPE html>
<html>
    <head></head>
    <body>
    <script type="text/javascript" src="/path/to/react.js"></script>
    <script type="text/javascript" src="/path/to/react-dom.js"></script>
    <script type="text/javascript">
        // Use react JavaScript code here or in a separate file
    </script>
    </body>
</html>
```

Um die JavaScript-Dateien zu erhalten, rufen Sie die Installationsseite der offiziellen React-Dokumentation auf.

React unterstützt auch die JSX-Syntax . JSX ist eine von Facebook erstellte Erweiterung, die JavaScript um XML-Syntax erweitert. Um JSX verwenden zu können, müssen Sie die Babel-Bibliothek einschließen und `<script type="text/javascript">` in `<script type="text/babel">` ändern, um JSX in Javascript-Code zu übersetzen.

```
<!DOCTYPE html>
<html>
    <head></head>
    <body>
    <script type="text/javascript" src="/path/to/react.js"></script>
    <script type="text/javascript" src="/path/to/react-dom.js"></script>
    <script src="https://npmcdn.com/babel-core@5.8.38/browser.min.js"></script>
    <script type="text/babel">
        // Use react JSX code here or in a separate file
    </script>
```

```
    </body>
</html>
```

Installation über npm

Sie können React auch mit npm installieren, indem Sie folgende Schritte ausführen :

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

Um React in Ihrem JavaScript-Projekt zu verwenden, haben Sie folgende Möglichkeiten:

```
var React = require('react');
var ReactDOM = require('react-dom');
ReactDOM.render(<App />, ...);
```

Installation über Garn

Facebook hat seinen eigenen Paketmanager namens Yarn veröffentlicht , mit dem auch React installiert werden kann. Nach der Installation von Yarn müssen Sie nur diesen Befehl ausführen:

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

Sie können React dann in Ihrem Projekt genauso verwenden, als wenn Sie React über npm installiert hätten.

Hallo Weltkomponente

Eine React-Komponente kann als eine ES6-Klasse definiert werden, die die Basisklasse `React.Component` . In seiner Minimalform, *muss* eine Komponente mit einer definieren `render` - Methode , die angibt , wie die Komponente auf das DOM macht. Die `render` gibt React-Knoten zurück, die mithilfe der JSX-Syntax als HTML-ähnliche Tags definiert werden können. Das folgende Beispiel zeigt, wie eine minimale Komponente definiert wird:

```
import React from 'react'

class HelloWorld extends React.Component {
    render() {
        return <h1>Hello, World!</h1>
    }
}

export default HelloWorld
```

Eine Komponente kann auch `props` erhalten. Dies sind Eigenschaften, die vom übergeordneten Element übergeben werden, um einige Werte anzugeben, die die Komponente nicht selbst kennen kann. Eine Eigenschaft kann auch eine Funktion enthalten, die von der Komponente aufgerufen werden kann, nachdem bestimmte Ereignisse `onClick` Beispielsweise könnte eine Schaltfläche eine Funktion für ihre `onClick` Eigenschaft `onClick` und bei jedem Klick aufrufen. Wenn Sie eine Komponente schreiben, können Sie auf ihre `props` über das `props` auf der Komponente selbst zugreifen:

```
import React from 'react'

class Hello extends React.Component {
    render() {
        return <h1>Hello, {this.props.name}!</h1>
    }
}

export default Hello
```

Das obige Beispiel zeigt , wie die Komponente kann eine beliebige Zeichenfolge in den vergangen machen `name` von ihrer Mutter prop. Beachten Sie, dass eine Komponente die empfangenen Requisiten nicht ändern kann.

Eine Komponente kann in einer anderen Komponente oder direkt in das DOM gerendert werden, wenn es sich um die oberste Komponente handelt. Verwenden Sie dazu `ReactDOM.render` und stellen Sie ihm sowohl die Komponente als auch den DOM-Knoten zur Verfügung, auf dem der React-Baum dargestellt werden soll:

```
import React from 'react'
import ReactDOM from 'react-dom'
import Hello from './Hello'

ReactDOM.render(<Hello name="Billy James" />, document.getElementById('main'))
```

Inzwischen wissen Sie, wie Sie eine grundlegende Komponente herstellen und `props` annehmen. Lasst uns noch einen Schritt weiter gehen und den `state` einführen.

Lassen Sie uns der Demo halber unsere Hello World-App erstellen und nur den Vornamen anzeigen, wenn ein vollständiger Name angegeben ist.

```
import React from 'react'

class Hello extends React.Component {

    constructor(props){

        //Since we are extending the default constructor,
        //handle default activities first.
        super(props);

        //Extract the first-name from the prop
        let firstName = this.props.name.split(" ")[0];

        //In the constructor, feel free to modify the
        //state property on the current context.
        this.state = {
            name: firstName
        }

    } //Look maa, no comma required in JSX based class defs!

    render() {
        return <h1>Hello, {this.state.name}!</h1>
    }
```

```
}

export default Hello
```

Hinweis: Jede Komponente kann ihren eigenen Status haben oder den Status ihrer Eltern als Requisite akzeptieren.

Codepen Link zum Beispiel.

Hallo Welt

Ohne JSX

Hier ein einfaches Beispiel, in dem die Haupt-API von React zum Erstellen eines React-Elements und die React-DOM-API zum Rendern des React-Elements im Browser verwendet wird.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello React!</title>

    <!-- Include the React and ReactDOM libraries -->
    <script src="https://fb.me/react-15.2.1.js"></script>
    <script src="https://fb.me/react-dom-15.2.1.js"></script>

  </head>
  <body>
    <div id="example"></div>

    <script type="text/javascript">

      // create a React element rElement
      var rElement = React.createElement('h1', null, 'Hello, world!');

      // dElement is a DOM container
      var dElement = document.getElementById('example');

      // render the React element in the DOM container
      ReactDOM.render(rElement, dElement);

    </script>

  </body>
</html>
```

Mit JSX

Anstatt ein React-Element aus Strings zu erstellen, können Sie JSX (eine von Facebook erstellte Javascript-Erweiterung zum Hinzufügen von XML-Syntax zu JavaScript) verwenden, die das Schreiben ermöglicht

```
var rElement = React.createElement('h1', null, 'Hello, world!');
```

als Äquivalent (und für jemanden, der mit HTML vertraut ist, besser lesbar)

```
var rElement = <h1>Hello, world!</h1>;
```

Der Code, der JSX enthält, muss in einem Tag `<script type="text/babel">` . Alles innerhalb dieses Tags wird mit der Babel-Bibliothek (die zusätzlich zu den React-Bibliotheken enthalten sein muss) in einfaches Javascript umgewandelt.

So wird schließlich das obige Beispiel:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello React!</title>

    <!-- Include the React and ReactDOM libraries -->
    <script src="https://fb.me/react-15.2.1.js"></script>
    <script src="https://fb.me/react-dom-15.2.1.js"></script>
    <!-- Include the Babel library -->
    <script src="https://npmcdn.com/babel-core@5.8.38/browser.min.js"></script>

  </head>
  <body>
   <div id="example"></div>

    <script type="text/babel">

      // create a React element rElement using JSX
      var rElement = <h1>Hello, world!</h1>;

      // dElement is a DOM container
      var dElement = document.getElementById('example');

      // render the React element in the DOM container
      ReactDOM.render(rElement, dElement);

    </script>

  </body>
</html>
```

Was ist ReactJS?

ReactJS ist eine komponentenbasierte Open-Source-Frontend-Bibliothek, die nur für die **Ansichtsebene** der Anwendung verantwortlich ist. Es wird von Facebook gepflegt.

ReactJS verwendet einen virtuellen DOM-basierten Mechanismus, um Daten (Ansichten) in HTML-DOM einzufügen. Das virtuelle DOM funktioniert schnell, da nur einzelne DOM-Elemente geändert werden, anstatt jedes Mal ein vollständiges DOM zu laden

> Eine React-Anwendung besteht aus mehreren **Komponenten** , von denen jede für die Ausgabe eines kleinen, wiederverwendbaren HTML-Teils verantwortlich ist. Komponenten können in andere Komponenten geschachtelt werden, damit komplexe

Anwendungen aus einfachen Bausteinen aufgebaut werden können. Eine Komponente kann auch den internen Zustand beibehalten. Beispielsweise kann eine TabList-Komponente eine Variable speichern, die der aktuell geöffneten Registerkarte entspricht.

Mit React können wir Komponenten in einer domänenspezifischen Sprache namens JSX schreiben. Mit JSX können wir unsere Komponenten mithilfe von HTML schreiben und JavaScript-Ereignisse einmischen. React konvertiert dies intern in ein virtuelles DOM und gibt schließlich unseren HTML-Code für uns aus.

React " *reagiert* " auf Zustandsänderungen in Ihren Komponenten schnell und automatisch, um die Komponenten im HTML-DOM mithilfe des virtuellen DOM erneut zu rendern. Das virtuelle DOM ist eine In-Memory-Darstellung eines tatsächlichen DOM. Durch den Großteil der Verarbeitung innerhalb des virtuellen DOMs und nicht direkt im DOM des Browsers kann React schnell reagieren und nur Komponenten hinzufügen, aktualisieren und entfernen, die seit dem letzten Renderzyklus geändert wurden.

Hallo Welt mit Staatenlosen Funktionen

Zustandslose Komponenten erhalten ihre Philosophie durch funktionale Programmierung. Was bedeutet: Eine Funktion gibt immer das gleiche wieder, genau das, was ihr gegeben wird.

Zum Beispiel:

```
const statelessSum = (a, b) => a + b;

let a = 0;
const statefulSum = () => a++;
```

Wie Sie im obigen Beispiel sehen können, gibt statelessSum immer die gleichen Werte zurück, wenn a und b angegeben sind. Die statefulSum-Funktion gibt jedoch nicht dieselben Werte zurück, auch wenn keine Parameter vorhanden sind. Dieses Funktionsverhalten wird auch als *Nebeneffekt bezeichnet* . Da wirkt sich die Komponente auf etwas weiter aus.

Es wird daher empfohlen, häufiger zustandslose Komponenten zu verwenden, da diese *frei von Nebenwirkungen sind* und immer dasselbe Verhalten erzeugen. Das ist es, was Sie in Ihren Apps suchen wollen, denn schwankender Zustand ist der ungünstigste Fall für ein wartbares Programm.

Die grundlegendste Art der reagierenden Komponente ist eine ohne Zustand. Reaktionskomponenten, die reine Funktionen ihrer Requisiten sind und keine interne Zustandsverwaltung erfordern, können als einfache JavaScript-Funktionen geschrieben werden. Diese werden als `Stateless Functional Components` da sie nur eine Funktion von `props` , ohne einen `state` zu haben, den man verfolgen kann.

Hier ist ein einfaches Beispiel, um das Konzept einer `Stateless Functional Component` zu veranschaulichen:

```
// In HTML
<div id="element"></div>

// In React
const MyComponent = props => {
    return <h1>Hello, {props.name}!</h1>;
};

ReactDOM.render(<MyComponent name="Arun" />, element);
// Will render <h1>Hello, Arun!</h1>
```

Beachten Sie, dass diese Komponente lediglich ein `h1` Element enthält, das den `name` prop enthält. Diese Komponente verfolgt keinen Zustand. Hier ist auch ein ES6-Beispiel:

```
import React from 'react'

const HelloWorld = props => (
    <h1>Hello, {props.name}!</h1>
)

HelloWorld.propTypes = {
    name: React.PropTypes.string.isRequired
}

export default HelloWorld
```

Da diese Komponenten keine Hintergrundinstanz zur Verwaltung des Status benötigen, bietet React mehr Raum für Optimierungen. Die Implementierung ist sauber, aber bisher wurden keine derartigen Optimierungen für zustandslose Komponenten implementiert .

React App erstellen

create -react-app ist ein von Facebook erstellter React-App-Boilerplate-Generator. Es bietet eine Entwicklungsumgebung, die für eine einfache Verwendung mit minimalem Setup konfiguriert ist, einschließlich:

- ES6- und JSX-Transpilation
- Dev-Server mit Hot-Modul-Nachladen
- Code fusseln
- CSS-Präfix
- Erstellen Sie ein Skript mit JS, CSS und Bildbündelung sowie Quellcaps
- Jest-Test-Framework

Installation

Installieren Sie zuerst create-react-app global mit dem Node Package Manager (npm).

```
npm install -g create-react-app
```

Führen Sie dann den Generator in Ihrem ausgewählten Verzeichnis aus.

```
create-react-app my-app
```

Navigieren Sie zu dem neu erstellten Verzeichnis und führen Sie das Startskript aus.

```
cd my-app/
npm start
```

Aufbau

Create-React-App ist absichtlich nicht standardmäßig konfigurierbar. Wenn eine nicht standardmäßige Verwendung erforderlich ist, um beispielsweise eine kompilierte CSS-Sprache wie Sass zu verwenden, kann der Befehl Eject verwendet werden.

```
npm run eject
```

Dies ermöglicht die Bearbeitung aller Konfigurationsdateien. NB: Dies ist ein irreversibler Prozess.

Alternativen

Alternative React Boilerplates umfassen:

- Enklave
- nwb
- Bewegung
- Rackt-Cli
- Budō
- rwb
- quik
- sagui
- roc

Build React App

Um Ihre App für die Produktionsbereitschaft zu erstellen, führen Sie den folgenden Befehl aus

```
npm run build
```

Absolute Grundlagen zum Erstellen wiederverwendbarer Komponenten

Komponenten und Requisiten

Da sich React nur mit der Sicht einer Anwendung befasst, wird der Hauptteil der Entwicklung in React die Erstellung von Komponenten sein. Eine Komponente repräsentiert einen Teil der Ansicht Ihrer Anwendung. "Requisiten" sind einfach die Attribute, die auf einem JSX-Knoten verwendet werden (z. B. `<SomeComponent someProp="some prop's value" />`) und sind die

hauptsächliche Art und Weise, wie unsere Anwendung mit unseren Komponenten interagiert. Im obigen Snippet hätten wir innerhalb von SomeComponent Zugriff auf `this.props` , dessen Wert das Objekt `{someProp: "some prop's value"}` .

Es kann hilfreich sein, React-Komponenten als einfache Funktionen zu verstehen - sie nehmen Eingaben in Form von "Requisiten" an und erzeugen Ausgaben als Markup. Viele einfache Komponenten gehen noch einen Schritt weiter und machen sich selbst zu "reinen Funktionen", was bedeutet, dass sie keine Nebeneffekte auslösen und idempotent sind (bei einer Reihe von Eingaben erzeugt die Komponente immer die gleiche Ausgabe). Dieses Ziel kann formal durchgesetzt werden, indem Komponenten anstelle von "Klassen" als Funktionen erstellt werden. Es gibt drei Möglichkeiten, eine React-Komponente zu erstellen:

- **Funktionale ("zustandslose") Komponenten**

```
const FirstComponent = props => (
    <div>{props.content}</div>
);
```

- **React.createClass ()**

```
const SecondComponent = React.createClass({
    render: function () {
        return (
            <div>{this.props.content}</div>
        );
    }
});
```

- **ES2015 Klassen**

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

Diese Komponenten werden genauso verwendet:

```
const ParentComponent = function (props) {
    const someText = "FooBar";
    return (
        <FirstComponent content={someText} />
        <SecondComponent content={someText} />
        <ThirdComponent content={someText} />
    );
}
```

Die obigen Beispiele werden alle zu identischem Markup führen.

Funktionale Komponenten können keinen "Zustand" enthalten. Wenn Ihre Komponente also einen

Status haben muss, sollten Sie sich für klassenbasierte Komponenten entscheiden. Weitere Informationen finden Sie unter Erstellen von Komponenten .

Als letzte Bemerkung sind reagierende Requisiten nach der Übergabe unveränderlich, dh sie können nicht innerhalb einer Komponente geändert werden. Wenn das übergeordnete Element einer Komponente den Wert einer Requisite ändert, behandelt React die alten Requisiten durch die neuen ersetzt, und die Komponente rendert sich selbst mit den neuen Werten.

In Thinking in React und wiederverwendbaren Komponenten finden Sie tiefere Einblicke in die Beziehung von Requisiten zu Komponenten.

Erste Schritte mit React online lesen: https://riptutorial.com/de/reactjs/topic/797/erste-schritte-mit-react

Kapitel 2: AJAX-Anruf reagieren

Examples

HTTP-GET-Anforderung

Manchmal muss eine Komponente einige Daten von einem entfernten Endpunkt (z. B. einer REST-API) wiedergeben. Eine Standardpraxis besteht darin, solche Aufrufe in der Methode `componentDidMount` durchzuführen.

Hier ist ein Beispiel, in dem Superagent als AJAX-Helfer verwendet wird:

```
import React from 'react'
import request from 'superagent'

class App extends React.Component {
  constructor () {
    super()
    this.state = {}
  }
  componentDidMount () {
    request
      .get('/search')
      .query({ query: 'Manny' })
      .query({ range: '1..5' })
      .query({ order: 'desc' })
      .set('API-Key', 'foobar')
      .set('Accept', 'application/json')
      .end((err, resp) => {
        if (!err) {
          this.setState({someData: resp.text})
        }
      })
  },
  render() {
    return (
      <div>{this.state.someData || 'waiting for response...'}</div>
    )
  }
}

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

Eine Anforderung kann initiiert werden, indem die entsprechende Methode für das `request` aufgerufen und anschließend `.end()` , um die Anforderung zu senden. Das Setzen von Headerfeldern ist einfach. `.set()` mit einem Feldnamen und `.set()` .

Die `.query()` -Methode akzeptiert Objekte, die bei Verwendung mit der GET-Methode eine `.query()` bilden. Im Folgenden wird der Pfad `/search?query=Manny&range=1..5&order=desc` .

POST- Anfragen

```
request.post('/user')
  .set('Content-Type', 'application/json')
  .send('{"name":"tj","pet":"tobi"}')
  .end(callback)
```

Weitere Informationen finden Sie unter Superagent-Dokumente .

Ajax in React ohne Bibliothek von Drittanbietern - auch bekannt mit VanillaJS

Folgendes würde in IE9 + funktionieren

```
import React from 'react'

class App extends React.Component {
  constructor () {
    super()
    this.state = {someData: null}
  }
  componentDidMount () {
    var request = new XMLHttpRequest();
    request.open('GET', '/my/url', true);

    request.onload = () => {
      if (request.status >= 200 && request.status < 400) {
        // Success!
        this.setState({someData: request.responseText})
      } else {
        // We reached our target server, but it returned an error
        // Possibly handle the error by changing your state.
      }
    };

    request.onerror = () => {
      // There was a connection error of some sort.
      // Possibly handle the error by changing your state.
    };

    request.send();
  },
  render() {
    return (
      <div>{this.state.someData || 'waiting for response...'}</div>
    )
  }
}

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

HTTP-GET-Anforderung und Durchschleifen von Daten

Das folgende Beispiel zeigt, wie ein aus einer Remote-Quelle erhaltener Datensatz in eine Komponente gerendert werden kann.

Wir erstellen eine AJAX-Anfrage mit `fetch` , das in die meisten Browser integriert ist. Verwenden Sie einen `fetch` polyfill in Produktion ältere Browser zu unterstützen. Sie können auch jede andere Bibliothek für Anfragen verwenden (z. B. Axios , SuperAgent oder sogar einfaches Javascript).

Die Daten, die wir erhalten, werden als Komponentenzustand festgelegt, sodass wir innerhalb der Render-Methode darauf zugreifen können. Dort durchlaufen wir die Daten mit einer `map` . Vergessen Sie nicht, dem Loop-Element immer ein eindeutiges `key` (oder eine Eigenschaft) hinzuzufügen, was für die Renderleistung von React wichtig ist.

```
import React from 'react';

class Users extends React.Component {
  constructor() {
    super();
    this.state = { users: [] };
  }

  componentDidMount() {
    fetch('/api/users')
      .then(response => response.json())
      .then(json => this.setState({ users: json.data }));
  }

  render() {
    return (
      <div>
        <h1>Users</h1>
        {
          this.state.users.length == 0
            ? 'Loading users...'
            : this.state.users.map(user => (
              <figure key={user.id}>
                <img src={user.avatar} />
                <figcaption>
                  {user.name}
                </figcaption>
              </figure>
            ))
        }
      </div>
    );
  }
}

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

Arbeitsbeispiel zu JSBin .

AJAX-Anruf reagieren online lesen: https://riptutorial.com/de/reactjs/topic/6432/ajax-anruf-reagieren

Kapitel 3: Boilerplate reagieren [Reagieren + Babel + Webpack]

Examples

Projekt einrichten

Sie benötigen Node Package Manager, um die Projektabhängigkeiten zu installieren. Laden Sie den Knoten für Ihr Betriebssystem von Nodejs.org herunter . Node Package Manager wird mit Node geliefert.

Sie können den Knotenversions- Manager auch verwenden, um Ihre Knoten- und NPM-Versionen besser zu verwalten. Es eignet sich hervorragend zum Testen Ihres Projekts auf verschiedenen Knotenversionen. Es wird jedoch nicht für die Produktionsumgebung empfohlen.

Wenn Sie Node auf Ihrem System installiert haben, installieren Sie einige wichtige Pakete, um Ihr erstes React-Projekt mit Babel und Webpack zu starten.

Bevor wir tatsächlich anfangen, Befehle im Terminal zu treffen. Schauen Sie sich an, wofür Babel und Webpack verwendet werden.

Sie können Ihr Projekt starten, indem Sie `npm init` in Ihrem Terminal `npm init` . Folgen Sie der Ersteinrichtung. Führen Sie danach die folgenden Befehle in Ihrem Terminal aus:

Abhängigkeiten:

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

Dev Dependecies:

```
npm install babel-core babel-loader babel-preset-es2015 babel-preset-react babel-preset-stage-0
webpack webpack-dev-server react-hot-loader --save-dev
```

Optionale Entwicklungsabhängigkeiten:

```
npm install eslint eslint-plugin-react babel-eslint --save-dev
```

Sie können sich auf dieses Beispiel package.json beziehen

Erstellen Sie `.babelrc` in Ihrem Projektstamm mit folgendem Inhalt:

```
{
  "presets": ["es2015", "stage-0", "react"]
}
```

Erstellen `.eslintrc` optional `.eslintrc` in Ihrem Projektstamm mit folgendem Inhalt:

```
{
```

```
  "ecmaFeatures": {
    "jsx": true,
    "modules": true
  },
  "env": {
    "browser": true,
    "node": true
  },
  "parser": "babel-eslint",
  "rules": {
    "quotes": [2, "single"],
    "strict": [2, "never"],
  },
  "plugins": [
    "react"
  ]
}
```

Erstellen Sie eine `.gitignore` Datei, um das Hochladen generierter Dateien in Ihr git-Repo zu verhindern.

```
node_modules
npm-debug.log
.DS_Store
dist
```

Erstellen `webpack.config.js` Datei `webpack.config.js` mit den folgenden Mindestinhalten.

```
var path = require('path');
var webpack = require('webpack');

module.exports = {
  devtool: 'eval',
  entry: [
    'webpack-dev-server/client?http://localhost:3000',
    'webpack/hot/only-dev-server',
    './src/index'
  ],
  output: {
    path: path.join(__dirname, 'dist'),
    filename: 'bundle.js',
    publicPath: '/static/'
  },
  plugins: [
    new webpack.HotModuleReplacementPlugin()
  ],
  module: {
    loaders: [{
      test: /\.js$/,
      loaders: ['react-hot', 'babel'],
      include: path.join(__dirname, 'src')
    }]
  }
};
```

Erstellen `sever.js` schließlich eine `sever.js` Datei, um `npm start` mit folgendem Inhalt `sever.js` zu können:

```
var webpack = require('webpack');
var WebpackDevServer = require('webpack-dev-server');
var config = require('./webpack.config');

new WebpackDevServer(webpack(config), {
  publicPath: config.output.publicPath,
  hot: true,
  historyApiFallback: true
}).listen(3000, 'localhost', function (err, result) {
  if (err) {
    return console.log(err);
  }

  console.log('Serving your awesome project at http://localhost:3000/');
});
```

Erstellen `src/app.js` Datei " `src/app.js` ", damit Ihr React-Projekt etwas `src/app.js` .

```
import React, { Component } from 'react';

export default class App extends Component {
  render() {
    return (
      <h1>Hello, world.</h1>
    );
  }
}
```

Führen Sie den `node server.js` oder `npm start` im Terminal aus, wenn Sie definiert haben, wofür `start` in `package.json`

Reaktiv-Starter-Projekt

Über dieses Projekt

Dies ist ein einfaches Boilerplate-Projekt. In diesem Beitrag erfahren Sie, wie Sie die Umgebung für ReactJs + Webpack + Bable einrichten.

Lass uns anfangen

Wir benötigen einen Knotenpaket-Manager, um den Express-Server hochzufahren und die Abhängigkeiten im gesamten Projekt zu verwalten. Wenn Sie mit Node Package Manager noch nicht vertraut sind, können Sie hier nachsehen. Hinweis: Die Installation des Knotenpaket-Managers ist hier erforderlich.

Erstellen Sie einen Ordner mit einem geeigneten Namen, und navigieren Sie vom Terminal oder über die GUI in diesen Ordner. Gehen Sie dann zum Terminal und geben Sie `npm init` Dies erstellt eine package.json-Datei. Nothing scary. Beschreibung, Einstiegspunkt, Git-Repository, Autor, Lizenz usw. Hier ist der Einstiegspunkt wichtig, da der Knoten beim Ausführen des Projekts zunächst danach sucht. Am Ende werden Sie aufgefordert, die von Ihnen bereitgestellten Informationen zu überprüfen. Sie können *Ja eingeben* oder ändern. Nun ist es *soweit* , unsere *package.json*- Datei ist fertig.

Express-Server-Setup *npm install express @ 4 --save ausführen* . Dies sind alle Abhängigkeiten, die wir für dieses Projekt benötigen. Hier ist das Save-Flag wichtig, ohne dass die *package.js*-Datei nicht aktualisiert wird. Die Hauptaufgabe von *package.json* ist das Speichern von Abhängigkeiten. Es wird die Expressversion 4 *hinzugefügt.* Ihr *package.json* wird wie `"dependencies": { "express": "^4.13.4", ..... },` aussehen `"dependencies": { "express": "^4.13.4", ..... },`

Nach dem vollständigen Download sehen Sie den Ordner *node_modules* und den Unterordner unserer Abhängigkeiten. Erstellen Sie nun im Stammverzeichnis des Projekts eine neue Datei *server.js* . Jetzt setzen wir den Express-Server. Ich werde den gesamten Code übergehen und es später erklären.

```
var express = require('express');
// Create our app
var app = express();

app.use(express.static('public'));

app.listen(3000, function () {
  console.log('Express server is using port:3000');
});
```

var express = erfordern ('express'); Dadurch erhalten Sie Zugriff auf die gesamte Express-API.

var app = express (); ruft die Express Library als Funktion auf. *app.use ();* Lassen Sie die Funktionalität Ihrer Express-Anwendung hinzufügen. *app.use (express.static ('public'));* gibt den Ordnernamen an, der auf unserem Webserver verfügbar gemacht wird. *app.listen (port, function () {})* wird hier *3000 sein,* und die von uns *aufgerufene Funktion* überprüft, ob der Web-Server ordnungsgemäß ausgeführt wird. Das ist es Express Server ist eingerichtet.

Gehen Sie nun zu unserem Projekt und erstellen Sie einen neuen öffentlichen Ordner und erstellen Sie die *index.html*- Datei. *index.html* ist die Standarddatei für Ihre Anwendung, und der Express-Server sucht nach dieser Datei. Die *index.html* ist eine einfache HTML-Datei, die aussieht

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="UTF-8"/>
</head>

<body>
  <h1>hello World</h1>
</body>

</html>
```

Gehen Sie durch das Terminal zum Projektpfad und geben Sie *node server.js ein* . Dann sehen Sie * console.log ('Express Server verwendet Port: 3000'); *.

Gehen Sie zum Browser und geben Sie *http: // localhost: 3000* in die Navigationsleiste ein. Sie sehen *Hallo Welt* .

Gehen Sie nun in den öffentlichen Ordner und erstellen Sie eine neue Datei *app.jsx* . JSX ist ein Präprozessor-Schritt, der JavaScript um XML-Syntax erweitert. Sie können React definitiv ohne JSX verwenden, aber JSX macht React um einiges eleganter. Hier ist der Beispielcode für *app.jsx*

```
ReactDOM.render(
  <h1>Hello World!!!</h1>,
  document.getElementById('app')
);
```

Gehen Sie nun zu *index.html* und ändern Sie den Code. Er sollte folgendermaßen aussehen

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="UTF-8"/>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23
/browser.min.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.7/react.js">
</script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.7/react-dom.js">    </script>
</head>

<body>
  <div id="app"></div>

  <script type="text/babel" src="app.jsx"></script>
</body>

</html>
```

Damit sind Sie fertig, ich hoffe, Sie finden es einfach.

Boilerplate reagieren [Reagieren + Babel + Webpack] online lesen: https://riptutorial.com/de/reactjs/topic/5969/boilerplate-reagieren--reagieren-plus-babel-plus-webpack-

Kapitel 4: Einführung in das serverseitige Rendering

Examples

Komponenten rendern

Es gibt zwei Optionen zum Rendern von Komponenten auf dem Server: `renderToString` und `renderToStaticMarkup` .

renderToString

Dadurch werden React-Komponenten in HTML auf dem Server dargestellt. Diese Funktion fügt den HTML-Elementen auch `data-react-` , sodass React on Client keine Elemente erneut darstellen muss.

```
import { renderToString } from "react-dom/server";
renderToString(<App />);
```

renderToStaticMarkup

Dadurch werden React-Komponenten als HTML- `data-react-` Eigenschaften von `data-react-` wird jedoch nicht empfohlen, Komponenten zu verwenden, die auf dem Client gerendert werden, da Komponenten neu gerendert werden.

```
import { renderToStaticMarkup } from "react-dom/server";
renderToStaticMarkup(<App />);
```

Einführung in das serverseitige Rendering online lesen:
https://riptutorial.com/de/reactjs/topic/7478/einfuhrung-in-das-serverseitige-rendering

Kapitel 5: Einrichten der React-Umgebung

Examples

Einfache Reaktionskomponente

Wir möchten die Komponente unten kompilieren und auf unserer Webseite darstellen können

Dateiname : src / index.jsx

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

class ToDo extends React.Component {
    render() {
        return (<div>I am working</div>);
    }
}

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

Installieren Sie alle Abhängigkeiten

```
# install react and react-dom
$ npm i react react-dom --save

# install webpack for bundling
$ npm i webpack -g

# install babel for module loading, bundling and transpiling
$ npm i babel-core babel-loader --save

# install babel presets for react and es6
$ npm i babel-preset-react babel-preset-es2015 --save
```

Konfigurieren Sie das Webpack

Erstellen Sie eine Datei `webpack.config.js` im Stammverzeichnis Ihres Arbeitsverzeichnisses

Dateiname : webpack.config.js

```
module.exports = {
    entry: __dirname + "/src/index.jsx",
    devtool: "source-map",
    output: {
        path: __dirname + "/build",
        filename: "bundle.js"
    },
    module: {
        loaders: [
```

```
            {test: /\.jsx?$/, exclude: /node_modules/, loader: "babel-loader"}
        ]
    }
}
```

Babel konfigurieren

Erstellen Sie eine Datei `.babelrc` im Stammverzeichnis unseres Arbeitsverzeichnisses

Dateiname : .babelrc

```
{
    "presets": ["es2015","react"]
}
```

HTML-Datei für die Reakt-Komponente

Richten Sie eine einfache HTML-Datei im Stammverzeichnis des Projektverzeichnisses ein

Dateiname : index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <div id="App"></div>
    <script src="build/bundle.js" charset="utf-8"></script>
  </body>
</html>
```

Transpile und bündeln Sie Ihre Komponente

Mit dem Webpack können Sie Ihre Komponente bündeln:

```
$ webpack
```

Dadurch wird unsere Ausgabedatei im `build` Verzeichnis erstellt.

Öffnen Sie die HTML-Seite in einem Browser, um die Komponente in Aktion anzuzeigen

Einrichten der React-Umgebung online lesen:
https://riptutorial.com/de/reactjs/topic/7480/einrichten-der-react-umgebung

Kapitel 6: Einrichten eines grundlegenden Webpacks, Reagieren und Babel-Umgebung

Bemerkungen

Diese Build-Pipeline ist nicht genau das, was Sie als "produktionsbereit" bezeichnen würden, aber es ist ein guter Anfang, um die Dinge hinzuzufügen, die Sie benötigen, um die gewünschte Entwicklungserfahrung zu erhalten. Einige Leute verfolgen (manchmal auch ich selbst) eine komplett aufgebaute Pipeline von Yeoman.io oder woanders und entfernen dann die Dinge, die sie nicht wollen, bis sie ihrem Stil entsprechen. Daran ist nichts falsch, aber vielleicht können Sie sich für das obige Beispiel für den umgekehrten Ansatz entscheiden und aus nackten Knochen aufbauen.

Einige Dinge, die Sie vielleicht hinzufügen möchten, sind Dinge wie ein Testframework und Abdeckungsstatistiken wie Karma mit Mocha oder Jasmin. Fusseln mit ESLint. Hot-Modul-Austausch im Webpack-Dev-Server, damit Sie diese Erfahrung mit Ctrl + S, F5 machen können. Außerdem wird die aktuelle Pipeline nur im dev-Modus erstellt, sodass eine Produktionsaufgabe gut ist.

Gotchas!

Beachten Sie in der context `webpack.config.js` der `webpack.config.js` wir den Pfad `webpack.config.js` des `webpack.config.js` definiert haben, anstatt `__dirname` mit der Zeichenfolge `/src` verketten . Um die Lösung plattformübergreifender zu machen, verwenden Sie den Hebelknoten, um uns zu helfen.

Erläuterung der Eigenschaften von webpack.config.js

Kontext

Dies ist der Dateipfad, für den Webpack als Stammpfad zum Auflösen relativer Dateipfade verwendet wird. In index.jsx, wo wir „request `require('./index.html')` , wird der Punkt tatsächlich in das Verzeichnis `src/` da wir ihn in dieser Eigenschaft als solche definiert haben.

Eintrag

Wo Webpack an erster Stelle steht, beginnt die Lösung. Deshalb sehen Sie in der index.jsx, dass wir die Lösung mit Anforderungen und Importen zusammennähen.

Ausgabe

Hier definieren wir, wo das Webpack die Dateien ablegen soll, die es als Bundle gefunden hat. Wir haben auch einen Namen für die Datei definiert, in der unser gebündeltes Javascript und unsere Styles gelöscht werden.

devServer

Dies sind spezifische Einstellungen für den Webpack-Dev-Server. Die `contentBase` legt fest, wo der Server sein root sein soll. Wir haben hier den Ordner `dist/` als Basis definiert. Der `port` ist der Port, auf dem der Server gehostet wird. `open` wird der webpack-dev-server angewiesen, Ihren Standardbrowser für Sie zu öffnen, sobald der Server hochgefahren ist.

Modul> Lader

Dies definiert ein Mapping, das Webpack verwenden soll, sodass es weiß, was zu tun ist, wenn andere Dateien gefunden werden. Die `test` -Eigenschaft gibt regex für webpack an, um zu bestimmen, ob es dieses Modul anwenden soll. In den meisten Fällen haben wir Übereinstimmungen mit Dateierweiterungen. `loader` oder `loaders` geben den Namen des Loader-Moduls an, mit dem wir die Datei in das Webpack laden möchten, und sorgen dafür, dass der Loader die Bündelung dieses Dateityps übernimmt. Es gibt auch eine `query` Eigenschaft im Javascript, diese enthält lediglich eine Abfragezeichenfolge für den Loader. Daher hätten wir wahrscheinlich auch eine Query-Eigenschaft für den HTML-Loader verwenden können, wenn wir dies wollten. Es ist nur eine andere Art, Dinge zu tun.

Examples

Erstellen einer Pipeline für eine angepasste "Hallo Welt" mit Bildern.

Schritt 1: Installieren Sie Node.js

Die Build-Pipeline, die Sie erstellen, basiert auf Node.js. Sie müssen also zunächst sicherstellen, dass Sie diese installiert haben. Eine Anleitung, wie installieren Node.js Sie die SO - Dokumentation für diese Prüfung können hier

Schritt 2: Initialisieren Sie Ihr Projekt als Knotenmodul

Öffnen Sie Ihren Projektordner in der Befehlszeile und verwenden Sie den folgenden Befehl:

```
npm init
```

Für die Zwecke dieses Beispiels können Sie die Standardeinstellungen übernehmen oder wenn Sie weitere Informationen dazu benötigen, was dies bedeutet, können Sie dieses SO-Dokument beim Einrichten der Paketkonfiguration überprüfen.

Schritt 3: Installieren Sie die erforderlichen npm-Pakete

Führen Sie den folgenden Befehl in der Befehlszeile aus, um die für dieses Beispiel erforderlichen Pakete zu installieren:

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

Dann für die Dev-Abhängigkeiten diesen Befehl ausführen:

```
npm install --save-dev babel-core babel-preset-react babel-preset-es2015 webpack babel-loader
css-loader style-loader file-loader image-webpack-loader
```

Schließlich sind webpack und webpack-dev-server Dinge, die es wert sind, global installiert zu werden, und nicht als Abhängigkeit von Ihrem Projekt. Wenn Sie es lieber als Abhängigkeit hinzufügen möchten, funktioniert das, ich tue es nicht. Hier ist der Befehl zum Ausführen:

```
npm install --global webpack webpack-dev-server
```

Schritt 3: Fügen Sie dem Stamm Ihres Projekts eine .babelrc-Datei hinzu

Dadurch wird Babel so eingerichtet, dass die gerade installierten Voreinstellungen verwendet werden. Ihre .babelrc-Datei sollte folgendermaßen aussehen:

```
{
  "presets": ["react", "es2015"]
}
```

Schritt 4: Projektverzeichnisstruktur einrichten

Richten Sie sich eine Verzeichnisstruktur ein, die im Stammverzeichnis Ihres Verzeichnisses wie folgt aussieht:

```
|- node_modules
|- src/
  |- components/
  |- images/
  |- styles/
  |- index.html
  |- index.jsx
|- .babelrc
|- package.json
```

HINWEIS: Die `node_modules` , `.babelrc` und `package.json` sollten bereits in den vorherigen Schritten vorhanden gewesen sein. Ich habe sie nur `node_modules` , damit Sie sehen können, wo sie passen.

Schritt 5: Füllen Sie das Projekt mit den Hello World-Projektdateien

Dies ist nicht wirklich wichtig für den Aufbau einer Pipeline. Ich gebe Ihnen einfach den Code für diese und Sie können sie kopieren und einfügen:

src / components / HelloWorldComponent.jsx

```
import React, { Component } from 'react';

class HelloWorldComponent extends Component {
  constructor(props) {
    super(props);
    this.state = {name: 'Student'};
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange(e) {
    this.setState({name: e.target.value});
  }
```

```
  render() {
    return (
      <div>
        <div className="image-container">
          <img src="./images/myImage.gif" />
        </div>
        <div className="form">
          <input type="text" onChange={this.handleChange} />
          <div>
            My name is {this.state.name} and I'm a clever cloggs because I built a React build
pipeline
          </div>
        </div>
      </div>
    );
  }
}

export default HelloWorldComponent;
```

src / images / meinImage.gif

Fühlen Sie sich frei, dies durch jedes Bild zu ersetzen, das Sie möchten, es ist einfach da, um zu beweisen, dass wir auch Bilder bündeln können. Wenn Sie Ihr eigenes Bild angeben und es anders benennen, müssen Sie die `HelloWorldComponent.jsx` aktualisieren, um Ihre Änderungen widerzuspiegeln. Ebenso , wenn Sie ein Bild mit einer anderen Dateierweiterung wählen , dann müssen Sie das ändern `test` Eigenschaft des Bildes Lader in den webpack.config.js mit entsprechenden regulären Ausdruck Ihrer neuen Dateierweiterung übereinstimmen ..

src / styles / styles.css

```
.form {
  margin: 25px;
  padding: 25px;
  border: 1px solid #ddd;
  background-color: #eaeaea;
  border-radius: 10px;
}

.form div {
  padding-top: 25px;
}

.image-container {
  display: flex;
  justify-content: center;
}
```

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Learning to build a react pipeline</title>
</head>
```

```
<body>
  <div id="content"></div>
  <script src="app.js"></script>
</body>
</html>
```

index.jsx

```
import React from 'react';
import { render } from 'react-dom';
import HelloWorldComponent from './components/HelloWorldComponent.jsx';

require('./images/myImage.gif');
require('./styles/styles.css');
require('./index.html');

render(<HelloWorldComponent />, document.getElementById('content'));
```

Schritt 6: Erstellen Sie eine Webpack-Konfiguration

Erstellen Sie eine Datei mit dem Namen webpack.config.js im Stammverzeichnis Ihres Projekts, und kopieren Sie diesen Code hinein:

webpack.config.js

```
var path = require('path');

var config = {
  context: path.resolve(__dirname + '/src'),
  entry: './index.jsx',
  output: {
    filename: 'app.js',
    path: path.resolve(__dirname + '/dist'),
  },
  devServer: {
    contentBase: path.join(__dirname + '/dist'),
    port: 3000,
    open: true,
  },
  module: {
    loaders: [
      {
        test: /\.(js|jsx)$/,
        exclude: /node_modules/,
        loader: 'babel-loader'
      },
      {
        test: /\.css$/,
        loader: "style!css"
      },
      {
        test: /\.gif$/,
        loaders: [
          'file?name=[path][name].[ext]',
          'image-webpack',
        ]
      },
      { test: /\.(html)$/,
```

```
        loader: "file?name=[path][name].[ext]"
      }
    ],
  },
};

module.exports = config;
```

Schritt 7: Erstellen Sie npm-Aufgaben für Ihre Pipeline

Dazu müssen Sie dem Scripts-Schlüssel des JSON, der in der Datei package.json im Stammverzeichnis Ihres Projekts definiert ist, zwei Eigenschaften hinzufügen. Machen Sie Ihren Skriptschlüssel so aussehen:

```
  "scripts": {
    "start": "webpack-dev-server",
    "build": "webpack",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
```

Das `test` bereits vorhanden und Sie können auswählen, ob Sie es behalten möchten oder nicht. Dies ist für dieses Beispiel nicht wichtig.

Schritt 8: Verwenden Sie die Pipeline

Wenn Sie sich in der Befehlszeile im Projektstammverzeichnis befinden, sollten Sie jetzt den Befehl ausführen können:

```
npm run build
```

Dadurch wird die von Ihnen erstellte kleine Anwendung gebündelt und in dem Verzeichnis `dist/` , das sie im Stammverzeichnis Ihres Projektordners erstellt.

Wenn Sie den Befehl ausführen:

```
npm start
```

Die von Ihnen erstellte Anwendung wird dann in Ihrem Standard-Webbrowser in einer Webpack-Dev-Serverinstanz bereitgestellt.

Einrichten eines grundlegenden Webpacks, Reagieren und Babel-Umgebung online lesen: https://riptutorial.com/de/reactjs/topic/6294/einrichten-eines-grundlegenden-webpacks--reagieren-und-babel-umgebung

Kapitel 7: Formulare und Benutzereingaben

Examples

Kontrollierte Komponenten

Kontrollierte Formularkomponenten werden mit einer `value` definiert. Der Wert der kontrollierten Eingaben wird von React verwaltet. Benutzereingaben haben keinen direkten Einfluss auf die gerenderten Eingaben. Eine Änderung der `value` Eigenschaft muss stattdessen diese Änderung widerspiegeln.

```
class Form extends React.Component {
  constructor(props) {
    super(props);

    this.onChange = this.onChange.bind(this);

    this.state = {
      name: ''
    };
  }

  onChange(e) {
    this.setState({
      name: e.target.value
    });
  }

  render() {
    return (
      <div>
        <label for='name-input'>Name: </label>
        <input
          id='name-input'
          onChange={this.onChange}
          value={this.state.name} />
      </div>
    )
  }
}
```

Das obige Beispiel zeigt, wie die `value` Eigenschaft den aktuellen Wert der Eingabe definiert und der `onChange` Ereignishandler den `onChange` der Komponente mit der Benutzereingabe aktualisiert.

Formulareingaben sollten nach Möglichkeit als kontrollierte Komponenten definiert werden. Dadurch wird sichergestellt, dass der Komponentenstatus und der Eingabewert jederzeit synchron sind, auch wenn der Wert durch einen anderen Trigger als eine Benutzereingabe geändert wird.

Unkontrollierte Komponenten

Unkontrollierte Komponenten - Eingänge, die keinen haben `value` Eigenschaft. Im Gegensatz zu kontrollierten Komponenten ist es die Aufgabe der Anwendung, den Komponentenstatus und den

Eingangswert synchron zu halten.

```
class Form extends React.Component {
  constructor(props) {
    super(props);

    this.onChange = this.onChange.bind(this);

    this.state = {
      name: 'John'
    };
  }

  onChange(e) {
    this.setState({
      name: e.target.value
    });
  }

  render() {
    return (
      <div>
        <label for='name-input'>Name: </label>
        <input
          id='name-input'
          onChange={this.onChange}
          defaultValue={this.state.name} />
      </div>
    )
  }
}
```

Hier wird der Status der Komponente ebenso wie bei kontrollierten Komponenten über den `onChange` Ereignishandler aktualisiert. Anstelle einer `value` Eigenschaft wird jedoch eine `defaultValue` Eigenschaft angegeben. Dies bestimmt den Anfangswert der Eingabe während des ersten Renderns. Nachfolgende Änderungen des Zustands der Komponente werden vom Eingabewert nicht automatisch übernommen. Wenn dies erforderlich ist, sollte stattdessen eine kontrollierte Komponente verwendet werden.

Formulare und Benutzereingaben online lesen:
https://riptutorial.com/de/reactjs/topic/2884/formulare-und-benutzereingaben

Kapitel 8: Installation

Examples

Einfaches Setup

Ordner einrichten

In diesem Beispiel wird davon `out/` dass sich der Code in `src/` und die Ausgabe in `out/` . Daher sollte die Ordnerstruktur ungefähr so aussehen

```
example/
|-- src/
|   |-- index.js
|   `-- ...
|-- out/
`-- package.json
```

Pakete einrichten

Unter der Annahme einer npm-Setup-Umgebung müssen wir zunächst babel einrichten, um den React-Code in einen es5-kompatiblen Code umzuwandeln.

```
$npm install --save-dev babel-core babel-loader babel-preset-es2015 babel-preset-react
```

Der obige Befehl weist npm an, die Core-Babel-Bibliotheken sowie das Loader-Modul für die Verwendung mit Webpack zu installieren. Wir installieren auch die es6 und reagieren auf die Voreinstellungen von babel, um den Code des JSX- und es6-Moduls zu verstehen. (Weitere Informationen zu den Presets finden Sie hier: Babel-Presets.)

```
$npm i -D webpack
```

Dieser Befehl installiert Webpack als Entwicklungsabhängigkeit. (**Ich** bin die Abkürzung für install und **-D** die Abkürzung für --save-dev)

Möglicherweise möchten Sie auch zusätzliche Webpack-Pakete installieren (z. B. zusätzliche Ladeprogramme oder die Erweiterung webpack-dev-server).

Zuletzt benötigen wir den eigentlichen Reaktionscode

```
$npm i -D react react-dom
```

Webpack einrichten

Bei der Einrichtung der Abhängigkeiten benötigen wir eine Datei webpack.config.js, um webpack mitzuteilen, was zu tun ist

einfache webpack.config.js:

```
var path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'out'),
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      {
        test: /\.js$/,
        exclude: /(node_modules)/,
        loader: 'babel-loader',
        query: {
          presets: ['es2015', 'react']
        }
      }
    ]
  }
};
```

Diese Datei teilt webpack mit der index.js Datei (angenommen, dass in src /) und wandeln sie in eine einzige bundle.js Datei im starten `out` - Verzeichnis.

Der `module` teilt webpack alle Dateien gegen den regulären Ausdruck angetroffen zu testen und wenn sie übereinstimmen, wird die angegebene Lader aufrufen. (`babel-loader` in diesem Fall) Darüber hinaus ist die `exclude` regex webpack erzählt für alle Module in der diese speziellen Lader zu ignorieren `node_modules` Ordner, hilft dies den transpilation Prozess zu beschleunigen. Schließlich informiert die `query` webpack, welche Parameter an babel übergeben werden, und wird verwendet, um die zuvor installierten Voreinstellungen weiterzugeben.

Setup testen

Jetzt müssen Sie nur noch die Datei `src/index.js` erstellen und die Anwendung packen

src / index.js:

```
'use strict'

import React from 'react'
import { render } from 'react-dom'

const App = () => {
  return <h1>Hello world!</h1>
}

render(
```

```
  <App />,
  document. getElementById('app')
)
```

Diese Datei würde normalerweise einen einfachen `<h1>Hello world!</h1>` -Header mit der ID 'app' in das html-Tag einfügen, aber für den Moment sollte es ausreichen, den Code einmal zu transpilieren.

`$./node_modules/.bin/webpack` . Führt die lokal installierte Version von Webpack aus (verwenden Sie `$webpack` wenn Sie Webpack global mit -g installiert haben)

Dies sollte die Datei `out/bundle.js` mit dem darin enthaltenen Code erstellen und schließt das Beispiel ab.

Webpack-dev-server verwenden

Konfiguration

Nachdem Sie ein einfaches Projekt für die Verwendung von Webpacks erstellt haben, wird Babel and Rea mit der Ausgabe von `$npm i -g webpack-dev-server` den Entwicklungs-HTTP-Server zur schnelleren Entwicklung installieren.

Webpack.config.js ändern

```
var path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'out'),
    publicPath: '/public/',
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      {
        test: /\.js$/,
        exclude: /(node_modules)/,
        loader: 'babel',
        query: {
          presets: ['es2015', 'react']
        }
      }
    ]
  },
  devServer: {
    contentBase: path.resolve(__dirname, 'public'),
    hot: true
  }
};
```

Die Modifikationen sind in

- `output.publicPath` der einen Pfad für die `output.publicPath` unseres Bundles festlegt (weitere Informationen finden Sie in den Webpack-Konfigurationsdateien)
- `devServer`
    - `contentBase` den `contentBase` für statische Dateien (zum Beispiel `index.html`)
    - `hot` setzt den webpack-dev-server auf hot reload, wenn Änderungen an Dateien auf der Festplatte vorgenommen werden

Und zum Schluss brauchen wir nur eine einfache index.html, um unsere App zu testen.

index.html:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>React Sandbox</title>
  </head>
  <body>

    <div id="app" />

    <script src="public/bundle.js"></script>
  </body>
</html>
```

Mit diesem Setup sollte `$webpack-dev-server` einen lokalen http-Server an Port 8080 starten und beim Verbinden eine Seite mit einer `<h1>Hello world!</h1>` rendern.

Installation online lesen: https://riptutorial.com/de/reactjs/topic/6441/installation

Kapitel 9: Installation von React, Webpack & Typescript

Bemerkungen

Um die Hervorhebung der Syntax in Ihrem Editor (z. B. VS-Code) zu erhalten, müssen Sie die Typinformationen für die Module herunterladen, die Sie in Ihrem Projekt verwenden.

Nehmen Sie beispielsweise an, Sie verwenden React und ReactDOM in Ihrem Projekt, und Sie möchten Hervorhebungen und Intellisense für sie erhalten. Sie müssen die Typen mit diesem Befehl zu Ihrem Projekt hinzufügen:

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

Ihr Editor sollte nun diese Typinformationen automatisch aufgreifen und Ihnen Autovervollständigung und Intellisense für diese Module zur Verfügung stellen.

Examples

webpack.config.js

```
module.exports = {
    entry: './src/index',
    output: {
        path: __dirname + '/build',
        filename: 'bundle.js'
    },
    module: {
        rules: [{
            test: /\.tsx?$/,
            loader: 'ts-loader',
            exclude: /node_modules/
        }]
    },
    resolve: {
        extensions: ['.ts', '.tsx']
    }
};
```

Die Hauptkomponenten sind (zusätzlich zu der Standard - `entry` , `output` und anderen Eigenschaften webpack):

Der Lader

Dazu müssen Sie eine Regel erstellen, die die `.tsx` `.ts` und `.tsx` prüft. `.tsx` als Lader `ts-loader` an.

TS-Erweiterungen auflösen

Sie müssen auch die Erweiterungen `.ts` und `.tsx` zum `resolve` hinzufügen, da das Webpack sie nicht erkennt.

tsconfig.json

Dies ist eine minimale Tsconfig, um Sie zum Laufen zu bringen.

```
{
    "include": [
        "src/*"
    ],
    "compilerOptions": {
        "target": "es5",
        "jsx": "react",
        "allowSyntheticDefaultImports": true
    }
}
```

Gehen wir die Eigenschaften einzeln durch:

`include`

Dies ist ein Array von Quellcode. Hier haben wir nur einen Eintrag, `src/*` , der angibt, dass alles im `src` Verzeichnis in die Kompilierung einbezogen werden soll.

`compilerOptions.target`

Gibt an, dass wir das Ziel auf ES5 kompilieren möchten

`compilerOptions.jsx`

Wenn Sie diese Option auf `true` , kompiliert TypeScript Ihre tsx-Syntax automatisch von `<div />` in `React.createElement("div")` .

`compilerOptions.allowSyntheticDefaultImports`

Praktische Eigenschaft, die es Ihnen ermöglicht, Knotenmodule als ES6-Module zu importieren

```
import * as React from 'react'
const { Component } = React
```

Sie können es einfach tun

```
import React, { Component } from 'react'
```

ohne Fehlermeldungen, dass React keinen Standardexport hat.

Meine erste Komponente

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';


interface AppProps {
    name: string;
}
interface AppState {
    words: string[];
}

class App extends Component<AppProps, AppState> {
    constructor() {
        super();
        this.state = {
            words: ['foo', 'bar']
        };
    }

    render() {
        const { name } = this.props;
        return (<h1>Hello {name}!</h1>);
    }
}

const root = document.getElementById('root');
ReactDOM.render(<App name="Foo Bar" />, root);
```

Wenn Sie TypeScript mit React verwenden, müssen Sie nach dem Herunterladen der Typdefinitionen von React DefinitelyTyped (`npm install --save @types/react`) `npm install --save @types/react` hinzufügen.

Das machst du gerne so:

```
class App extends Component<AppProps, AppState> { }
```

`AppProps` und `AppState` sind Schnittstellen (oder Typ-Aliasnamen) für die Requisiten und den Status Ihrer Komponenten.

Installation von React, Webpack & Typescript online lesen:
https://riptutorial.com/de/reactjs/topic/9590/installation-von-react--webpack--amp--typescript

Kapitel 10: JSX

Bemerkungen

JSX ist ein **Präprozessor-Schritt** , der JavaScript um XML-Syntax erweitert. Sie können React definitiv ohne JSX verwenden, aber JSX macht React viel eleganter.

Wie XML haben JSX-Tags einen Tagnamen, Attribute und Kinder. Wenn ein Attributwert in Anführungszeichen eingeschlossen ist, ist der Wert eine Zeichenfolge. Umschließen Sie den Wert andernfalls in geschweifte Klammern, und der Wert ist der eingeschlossene JavaScript-Ausdruck.

Grundsätzlich liefert JSX nur syntaktischen Zucker für die Funktion `React.createElement(component, props, ...children)` .

Also der folgende JSX-Code:

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

ReactDOM.render(<HelloMessage name="Kalo" />, mountNode);
```

Kompiliert sich mit dem folgenden JavaScript-Code:

```
class HelloMessage extends React.Component {
  render() {
    return React.createElement(
      "div",
      null,
      "Hello ",
      this.props.name
    );
  }
}

ReactDOM.render(React.createElement(HelloMessage, { name: "Kalo" }), mountNode);
```

Beachten Sie abschließend, dass **die folgende Zeile in JSX weder eine Zeichenfolge noch HTML ist** :

```
const element = <h1>Hello, world!</h1>;
```

Es heißt JSX und ist eine **Syntaxerweiterung für JavaScript** . JSX erinnert Sie möglicherweise an eine Vorlagensprache, verfügt jedoch über die volle Funktionalität von JavaScript.

Das React-Team sagt in seinen Dokumenten, dass es empfohlen wird, es zu verwenden, um zu beschreiben, wie die Benutzeroberfläche aussehen sollte.

Examples

Requisiten in JSX

Es gibt verschiedene Möglichkeiten, Requisiten in JSX anzugeben.

JavaScript-Ausdrücke

Sie können **jeden JavaScript-Ausdruck** als Requisite übergeben, indem Sie ihn mit `{}` . Zum Beispiel in dieser JSX:

```
<MyComponent count={1 + 2 + 3 + 4} />
```

In `MyComponent` der Wert von `props.count 10` , da der Ausdruck `1 + 2 + 3 + 4` ausgewertet wird.

Wenn Anweisungen und for-Schleifen keine Ausdrücke in JavaScript sind, können sie nicht direkt in JSX verwendet werden.

String Literals

Natürlich können Sie auch ein beliebiges `string literal` als `prop` . Diese beiden JSX-Ausdrücke sind gleichwertig:

```
<MyComponent message="hello world" />

<MyComponent message={'hello world'} />
```

Wenn Sie ein Zeichenfolgenliteral übergeben, hat der Wert keinen HTML-Code. Diese beiden JSX-Ausdrücke sind also gleichwertig:

```
<MyComponent message="&lt;3" />

<MyComponent message={'<3'} />
```

Dieses Verhalten ist normalerweise nicht relevant. Der Vollständigkeit halber wird es hier nur erwähnt.

Standardwert für Requisiten

Wenn Sie keinen Wert für eine Eigenschaft übergeben, **wird der Standardwert " `true` .** Diese beiden JSX-Ausdrücke sind gleichwertig:

```
<MyTextBox autocomplete />

<MyTextBox autocomplete={true} />
```

Allerdings sagt der React Team in ihre docs **diesem Ansatz wird nicht empfohlen,** weil es mit dem ES6 Objekt Stenografie verwechselt werden kann `{foo}` , die für kurz `{foo: foo}` anstatt `{foo: true}` . Sie sagen, dass dieses Verhalten gerade da ist, so dass es dem Verhalten von HTML entspricht.

Attribute verteilen

Wenn Sie bereits Requisiten als Objekt haben und Sie diese in JSX übergeben möchten, können Sie `...` als Spread-Operator verwenden, um das gesamte Requisitenobjekt zu übergeben. Diese beiden Komponenten sind gleichwertig:

```
function Case1() {
  return <Greeting firstName="Kaloyab" lastName="Kosev" />;
}

function Case2() {
  const person = {firstName: 'Kaloyan', lastName: 'Kosev'};
  return <Greeting {...person} />;
}
```

Kinder in JSX

In JSX-Ausdrücken, die sowohl ein öffnendes Tag als auch ein schließendes Tag enthalten, wird der Inhalt zwischen diesen Tags als spezielles prop: `props.children` . Es gibt verschiedene Möglichkeiten, Kinder zu übergeben:

String Literals

Sie können eine Zeichenfolge zwischen den öffnenden und schließenden Tags setzen, und `props.children` wird nur diese Zeichenfolge sein. Dies ist nützlich für viele der integrierten HTML-Elemente. Zum Beispiel:

```
<MyComponent>
    <h1>Hello world!</h1>
</MyComponent>
```

Dies ist gültiges JSX, und `props.children` in MyComponent wird einfach `<h1>Hello world!</h1>` .

Beachten Sie, dass **der HTML-Code nicht maskiert ist** , sodass Sie JSX im Allgemeinen genauso wie HTML schreiben können.

Bedenken Sie, dass in diesem Fall JSX:

- entfernt Leerzeichen am Anfang und Ende einer Zeile;
- entfernt leere Zeilen;
- neue Zeilen neben Tags werden entfernt;
- Neue Zeilen, die in der Mitte von String-Literalen auftreten, werden zu einem einzigen Raum zusammengefasst.

JSX-Kinder

Sie können mehr JSX-Elemente als untergeordnete Elemente angeben. Dies ist nützlich, um verschachtelte Komponenten anzuzeigen:

```
<MyContainer>
  <MyFirstComponent />
  <MySecondComponent />
</MyContainer>
```

Sie können **verschiedene Kindertypen zusammenmischen, sodass Sie String-Literale zusammen mit JSX-Kindern verwenden können** . Dies ist eine andere Art und Weise, in der JSX wie HTML ist, so dass dies sowohl gültiges JSX als auch gültiges HTML ist:

```
<div>
  <h2>Here is a list</h2>
  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
  </ul>
</div>
```

Beachten Sie, dass eine React-Komponente **nicht mehrere React-Elemente zurückgeben kann. Ein einzelner JSX-Ausdruck kann jedoch mehrere untergeordnete Elemente enthalten** . Wenn Sie möchten, dass eine Komponente mehrere Elemente rendert, können Sie sie wie im obigen Beispiel in ein `div` .

JavaScript-Ausdrücke

Sie können jeden JavaScript-Ausdruck als untergeordnete Elemente übergeben, indem Sie ihn in `{}` einschließen. Zum Beispiel sind diese Ausdrücke gleichwertig:

```
<MyComponent>foo</MyComponent>

<MyComponent>{'foo'}</MyComponent>
```

Dies ist oft nützlich, um eine Liste von JSX-Ausdrücken mit beliebiger Länge zu rendern. So wird beispielsweise eine HTML-Liste dargestellt:

```
const Item = ({ message }) => (
  <li>{ message }</li>
);

const TodoList = () => {
  const todos = ['finish doc', 'submit review', 'wait stackoverflow review'];
  return (
    <ul>
      { todos.map(message => (<Item key={message} message={message} />)) }
    </ul>
  );
};
```

Beachten Sie, dass JavaScript-Ausdrücke mit anderen Kindertypen gemischt werden können.

Funktionen als Kinder

Normalerweise werden in JSX eingefügte JavaScript-Ausdrücke zu einem String, einem React-Element oder einer Liste dieser Elemente ausgewertet. `props.children` funktioniert jedoch genauso wie alle anderen Requisiten, da sie jede Art von Daten übergeben können, nicht nur die, die React rendern kann. Wenn Sie beispielsweise eine benutzerdefinierte Komponente haben, können Sie einen Rückruf als `props.children` :

```
const ListOfTenThings = () => (
  <Repeat numTimes={10}>
    {(index) => <div key={index}>This is item {index} in the list</div>}
  </Repeat>
);

// Calls the children callback numTimes to produce a repeated component
const Repeat = ({ numTimes, children }) => {
  let items = [];
  for (let i = 0; i < numTimes; i++) {
    items.push(children(i));
  }
  return <div>{items}</div>;
};
```

Untergeordnete Elemente, die an eine benutzerdefinierte Komponente übergeben werden, können alles sein, solange sie von der Komponente in etwas umgewandelt werden, das React vor dem Rendern verstehen kann. Diese Verwendung ist nicht üblich, sie funktioniert jedoch, wenn Sie die Möglichkeiten von JSX erweitern möchten.

Ignorierte Werte

Beachten Sie, dass `false` , `null` , `undefined` und `true` gültige Kinder sind. Aber sie rendern einfach nicht. Diese JSX-Ausdrücke werden alle auf dieselbe Weise gerendert:

```
<MyComponent />

<MyComponent></MyComponent>

<MyComponent>{false}</MyComponent>

<MyComponent>{null}</MyComponent>

<MyComponent>{true}</MyComponent>
```

Dies ist äußerst nützlich, um React-Elemente bedingt zu rendern. Dieser JSX gibt nur ein aus, wenn `showHeader` wahr ist:

```
<div>
  {showHeader && <Header />}
  <Content />
</div>
```

Ein wichtiger Nachteil ist, dass einige "falsche" Werte wie die `0` Zahl immer noch von React gerendert werden. Dieser Code verhält sich beispielsweise nicht wie erwartet, da `0` gedruckt wird, wenn `props.messages` ein leeres Array ist:

```
<div>
  {props.messages.length &&
    <MessageList messages={props.messages} />
  }
</div>
```

Um dies zu beheben, stellen Sie sicher, dass der Ausdruck vor dem `&&` immer boolean ist:

```
<div>
  {props.messages.length > 0 &&
    <MessageList messages={props.messages} />
  }
</div>
```

Beachten Sie schließlich, dass Sie einen Wert wie `false` , `true` , `null` oder `undefined` in der Ausgabe anzeigen möchten, wenn Sie ihn in einen String konvertieren müssen:

```
<div>
  My JavaScript variable is {String(myVariable)}.
</div>
```

JSX online lesen: https://riptutorial.com/de/reactjs/topic/8027/jsx

Kapitel 11: Kommunikation zwischen Komponenten

Bemerkungen

Es gibt insgesamt drei Kommunikationsfälle zwischen den React-Komponenten:

- Fall 1: Kommunikation zwischen Eltern und Kindern
- Fall 2: Kommunikation von Kind zu Eltern
- Fall 3: Kommunikation der Komponenten (keine Komponente zu jeder Komponente)

Examples

Übergeordnete zu untergeordneten Komponenten

Dass der einfachste Fall in der React-Welt sehr natürlich ist und die Chancen stehen - Sie nutzen ihn bereits.

Sie können **Requisiten an untergeordnete Komponenten übergeben** . In diesem Beispiel `message` ist die Stütze , die wir an die untergeordnete Komponente überliefern , wird der Name Nachricht willkürlich gewählt, können Sie es nennen wie Sie wollen.

```
import React from 'react';

class Parent extends React.Component {
    render() {
        const variable = 5;
        return (
            <div>
                <Child message="message for child" />
                <Child message={variable} />
            </div>
        );
    }
}

class Child extends React.Component {
    render() {
        return <h1>{this.props.message}</h1>
    }
}

export default Parent;
```

Hier stellt die `<Parent />` -Komponente zwei `<Child />` -Komponenten dar, wobei die `message for child` in der ersten Komponente und `5` in der zweiten Komponente übergeben werden.

Zusammenfassend haben Sie eine Komponente (übergeordnete Komponente), die eine andere Komponente (untergeordnete Komponente) wiedergibt und einige Requisiten an sie übergibt.

Kind zu übergeordneten Komponenten

Wenn Sie Daten an das übergeordnete Element zurücksenden, übergeben wir dazu einfach **eine Funktion von der übergeordneten Komponente an die untergeordnete Komponente** , und **die untergeordnete Komponente ruft diese Funktion auf** .

In diesem Beispiel ändern wir den übergeordneten Status, indem Sie eine Funktion an die untergeordnete Komponente übergeben und diese Funktion innerhalb der untergeordneten Komponente aufrufen.

```
import React from 'react';

class Parent extends React.Component {
    constructor(props) {
        super(props);
        this.state = { count: 0 };

        this.outputEvent = this.outputEvent.bind(this);
    }
    outputEvent(event) {
        // the event context comes from the Child
        this.setState({ count: this.state.count++ });
    }

    render() {
        const variable = 5;
        return (
            <div>
                Count: { this.state.count }
                <Child clickHandler={this.outputEvent} />
            </div>
        );
    }
}

class Child extends React.Component {
    render() {
        return (
            <button onClick={this.props.clickHandler}>
                Add One More
            </button>
        );
    }
}

export default Parent;
```

Beachten Sie, dass die `outputEvent` Methode des Parent (die den Parent- `outputEvent` ändert) vom `onClick` Ereignis der Schaltfläche des `onClick` aufgerufen wird.

Nicht verwandte Komponenten

Die einzige Möglichkeit, wenn Ihre Komponenten keine Eltern-Kind-Beziehung haben (oder verwandt sind, aber zu weiter fortgeschritten sind, wie z. B. ein Grand-Grand-Sohn), besteht darin, ein Signal zu haben, das eine Komponente abonniert und in die andere geschrieben wird.

Dies sind die zwei Grundoperationen eines Ereignissystems: **Abonnieren / Abhören** eines Ereignisses, das benachrichtigt werden soll, und **Senden / Auslösen / Veröffentlichen / Senden** eines Ereignisses, um diejenigen zu benachrichtigen, die dies wünschen.

Dafür gibt es mindestens 3 Muster. Einen Vergleich finden Sie hier .

Hier ist eine kurze Zusammenfassung:

- Muster 1: **Ereignissender / Ziel / Dispatcher** : Die Zuhörer müssen auf die Quelle verweisen, um sie zu abonnieren.
    - Abonnieren: `otherObject.addEventListener('click', () => { alert('click!'); });`
    - zu versenden: `this.dispatchEvent('click');`
- Muster 2: **Veröffentlichen / Abonnieren** : Sie benötigen keinen bestimmten Verweis auf die Quelle, die das Ereignis auslöst. Es gibt ein globales Objekt, auf das alle Ereignisse zugreifen können.
    - Abonnieren: `globalBroadcaster.subscribe('click', () => { alert('click!'); });`
    - zu `globalBroadcaster.publish('click');` : `globalBroadcaster.publish('click');`
- Muster 3: **Signale** : Ähnlich wie Event Emitter / Target / Dispatcher, aber Sie verwenden hier keine zufälligen Zeichenfolgen. Jedes Objekt, das Ereignisse ausgeben kann, muss eine bestimmte Eigenschaft mit diesem Namen haben. Auf diese Weise wissen Sie genau, welche Ereignisse ein Objekt ausgeben kann.
    - Abonnieren: `otherObject.clicked.add( () => { alert('click'); });`
    - zu `this.clicked.dispatch();` : `this.clicked.dispatch();`

Kommunikation zwischen Komponenten online lesen:
https://riptutorial.com/de/reactjs/topic/6567/kommunikation-zwischen-komponenten

Kapitel 12: Kommunizieren Sie zwischen Komponenten

Examples

Kommunikation zwischen zustandslosen Funktionskomponenten

In diesem Beispiel werden `Redux` und `React Redux` Module verwendet, um den Anwendungsstatus zu handhaben und die Funktionskomponenten automatisch wiederzugeben., Und natürlich `React` und `React Dom`

Sie können die vollständige Demo hier abschließen

Im folgenden Beispiel haben wir drei verschiedene Komponenten und eine verbundene Komponente

- **UserInputForm** : Diese Komponente zeigt ein Eingabefeld an. Wenn sich der `inputChange` ändert, wird die `inputChange` Methode für `props` (die von der übergeordneten Komponente bereitgestellt werden) `inputChange` Wenn die Daten ebenfalls angegeben werden, wird dies im Eingabefeld angezeigt.
- **UserDashboard** : Diese Komponente zeigt eine einfache Nachricht an und `UserInputForm` Komponente. Außerdem übergibt sie die `inputChange` Methode an die `UserInputForm` Komponente. Die `UserInputForm` Komponente inturn verwendet diese Methode zur Kommunikation mit der übergeordneten Komponente.
    - **UserDashboardConnected** : Diese Komponente `UserDashboard` die `UserDashboard` Komponente `UserDashboard` mit der `ReactRedux connect` . Dies erleichtert uns die Verwaltung des Komponentenstatus und das Aktualisieren der Komponente, wenn sich der Status ändert.
- **App** : Diese Komponente rendert nur die `UserDashboardConnected` Komponente.

```
const UserInputForm = (props) => {

  let handleSubmit = (e) => {
    e.preventDefault();
  }

  return(
    <form action="" onSubmit={handleSubmit}>
      <label htmlFor="name">Please enter your name</label>
      <br />
      <input type="text" id="name" defaultValue={props.data.name || ''} onChange={
props.inputChange } />
    </form>
  )

}
```

```
const UserDashboard = (props) => {

  let inputChangeHandler = (event) => {
    props.updateName(event.target.value);
  }

  return(
    <div>
      <h1>Hi { props.user.name || 'User' }</h1>
      <UserInputForm data={props.user} inputChange={inputChangeHandler} />
    </div>
  )

}

const mapStateToProps = (state) => {
  return {
    user: state
  };
}
const mapDispatchToProps = (dispatch) => {
  return {
    updateName: (data) => dispatch( Action.updateName(data) ),
  };
};

const { connect, Provider } = ReactRedux;
const UserDashboardConnected = connect(
  mapStateToProps,
  mapDispatchToProps
)(UserDashboard);



const App = (props) => {
  return(
    <div>
      <h1>Communication between Stateless Functional Components</h1>
      <UserDashboardConnected />
    </div>
  )
}



const user = (state={name: 'John'}, action) => {
  switch (action.type) {
    case 'UPDATE_NAME':
      return Object.assign( {}, state, {name: action.payload}  );

    default:
      return state;
  }
};

const { createStore } = Redux;
const store = createStore(user);
const Action = {
  updateName: (data) => {
```

```
    return { type : 'UPDATE_NAME', payload: data }
  },
}


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

JS-Bin-URL

Kommunizieren Sie zwischen Komponenten online lesen:
https://riptutorial.com/de/reactjs/topic/6137/kommunizieren-sie-zwischen-komponenten

Kapitel 13: Komponenten

Bemerkungen

`React.createClass` wurde in Version 15.5 nicht mehr unterstützt und wird voraussichtlich in Version 16 entfernt . Für diejenigen, die es noch benötigen, gibt es ein Drop-In-Ersatzpaket . Beispiele, die es verwenden, sollten aktualisiert werden.

Examples

Basiskomponente

Gegeben die folgende HTML-Datei:

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>React Tutorial</title>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react-dom.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-
core/5.8.34/browser.min.js"></script>
  </head>
  <body>
    <div id="content"></div>
    <script type="text/babel" src="scripts/example.js"></script>
  </body>
</html>
```

Sie können eine Basiskomponente mit dem folgenden Code in einer separaten Datei erstellen:

scripts / example.js

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';

class FirstComponent extends Component {
  render() {
    return (
      <div className="firstComponent">
        Hello, world! I am a FirstComponent.
      </div>
    );
  }
}
ReactDOM.render(
  <FirstComponent />, // Note that this is the same as the variable you stored above
  document.getElementById('content')
);
```

Sie erhalten folgendes Ergebnis (beachten Sie, was sich im `div#content` von `div#content`):

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>React Tutorial</title>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react-dom.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-
core/5.8.34/browser.min.js"></script>
  </head>
  <body>
    <div id="content">
        <div className="firstComponent">
            Hello, world! I am a FirstComponent.
        </div>
    </div>
    <script type="text/babel" src="scripts/example.js"></script>
  </body>
</html>
```

Komponenten verschachteln

Die Stärke von ReactJS liegt in der Fähigkeit, Komponenten verschachteln zu können. Nehmen Sie die folgenden zwei Komponenten:

```
var React = require('react');
var createReactClass = require('create-react-class');

var CommentList = reactCreateClass({
  render: function() {
    return (
      <div className="commentList">
        Hello, world! I am a CommentList.
      </div>
    );
  }
});

var CommentForm = reactCreateClass({
  render: function() {
    return (
      <div className="commentForm">
        Hello, world! I am a CommentForm.
      </div>
    );
  }
});
```

Sie können diese Komponenten in der Definition einer anderen Komponente verschachteln und darauf verweisen:

```
var React = require('react');
var createReactClass = require('create-react-class');
```

```
var CommentBox = reactCreateClass({
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList /> // Which was defined above and can be reused
        <CommentForm /> // Same here
      </div>
    );
  }
});
```

Die weitere Verschachtelung kann auf drei Arten erfolgen, die alle ihre eigenen Plätze haben.

1. Schachteln ohne Kinder

(Fortsetzung von oben)

```
var CommentList = reactCreateClass({
  render: function() {
    return (
      <div className="commentList">
        <ListTitle/>
        Hello, world! I am a CommentList.
      </div>
    );
  }
});
```

Dies ist der Stil, in dem A B und B C komponiert.

Pros

- Einfache und schnelle Trennung von UI-Elementen
- Je nach Zustand der übergeordneten Komponente lassen sich Requisiten einfach für Kinder injizieren

Cons

- Weniger Einsicht in die Kompositionsarchitektur
- Weniger Wiederverwendbarkeit

Gut wenn

- B und C sind nur Präsentationskomponenten
- B sollte für den Lebenszyklus von C verantwortlich sein

2. Verschachtelung mit Kindern

(Fortsetzung von oben)

```
var CommentBox = reactCreateClass({
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList>
            <ListTitle/> // child
        </CommentList>
        <CommentForm />
      </div>
    );
  }
});
```

Dies ist der Stil, in dem A aus B besteht und A B zu B anweist.

Pros

- Besseres Komponentenlebenszyklusmanagement
- Bessere Sichtbarkeit der Kompositionsarchitektur
- Bessere Wiederverwendbarkeit

Cons

- Das Einspritzen von Requisiten kann etwas teuer werden
- Weniger Flexibilität und Kraft in untergeordneten Komponenten

Gut wenn

- B sollte in der Zukunft oder anderswo etwas anderes als C komponieren
- A sollte den Lebenszyklus von C steuern

B würde C mit `this.props.children` rendern, und es gibt keinen strukturierten Weg für B, um `this.props.children` , wozu diese Kinder dienen. So kann B die untergeordneten Komponenten durch das Hinzufügen zusätzlicher Requisiten bereichern, aber wenn B genau wissen muss, was sie sind, ist # 3 möglicherweise die bessere Option.

3. Verschachtelung mit Requisiten

(Fortsetzung von oben)

```
var CommentBox = reactCreateClass({
```

```
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList title={ListTitle}/> //prop
        <CommentForm />
      </div>
    );
  }
});
```

Dies ist der Stil, in dem A B und B zusammenstellt und A die Möglichkeit bietet, etwas zu übergeben, um es zu einem bestimmten Zweck zu komponieren. Strukturiertere Komposition.

Pros

- Komposition als Merkmal
- Einfache Validierung
- Bessere Komposaiblilität

Cons

- Das Einspritzen von Requisiten kann etwas teuer werden
- Weniger Flexibilität und Kraft in untergeordneten Komponenten

Gut wenn

- In B sind bestimmte Merkmale definiert, um etwas zu komponieren
- B sollte nur wissen, was gerendert werden soll

Nummer 3 ist in der Regel ein Muss, um eine öffentliche Bibliothek mit Komponenten zu erstellen. Im Allgemeinen ist es jedoch eine gute Praxis, komponierbare Komponenten zu erstellen und die Kompositionsmerkmale klar zu definieren. # 1 ist am einfachsten und schnellsten, um etwas zu schaffen, das funktioniert, aber # 2 und # 3 sollten in verschiedenen Anwendungsfällen gewisse Vorteile bieten.

Komponenten erstellen

Dies ist eine Erweiterung von Basic Example:

Grundstruktur

```
import React, { Component } from 'react';
import { render } from 'react-dom';

class FirstComponent extends Component {
    render() {
```

```
        return (
            <div>
                Hello, {this.props.name}! I am a FirstComponent.
            </div>
        );
    }
}

render(
    <FirstComponent name={ 'User' } />,
    document.getElementById('content')
);
```

Das obige Beispiel wird als **zustandslose** Komponente bezeichnet, da es keinen Zustand enthält (im Sinne des Wortes React).

In einem solchen Fall ist es für manche Leute empfehlenswert, Stateless Functional Components zu verwenden, die auf ES6-Pfeilfunktionen basieren.

Zustandslose Funktionskomponenten

In vielen Anwendungen gibt es intelligente Komponenten, die Status enthalten, aber dumme Komponenten darstellen, die einfach Requisiten empfangen und HTML als JSX zurückgeben. Zustandslose Funktionskomponenten sind viel wiederverwendbar und wirken sich positiv auf Ihre Anwendung aus.

Sie haben 2 Hauptmerkmale:

1. Wenn sie gerendert werden, erhalten sie ein Objekt mit allen übergebenen Requisiten
2. Sie müssen den JSX zurückgeben, um gerendert zu werden

```
// When using JSX inside a module you must import React
import React from 'react';
import PropTypes from 'prop-types';

const FirstComponent = props => (
    <div>
        Hello, {props.name}! I am a FirstComponent.
    </div>
);

//arrow components also may have props validation
FirstComponent.propTypes = {
    name: PropTypes.string.isRequired,
}

// To use FirstComponent in another file it must be exposed through an export call:
export default FirstComponent;
```

Stateful-Komponenten

Im Gegensatz zu den oben gezeigten 'statuslosen' Komponenten weisen 'stateful'-Komponenten ein `setState` , das mit der `setState` Methode aktualisiert werden kann. Der Status muss im `constructor` initialisiert werden, bevor er gesetzt werden kann:

```
import React, { Component } from 'react';

class SecondComponent extends Component {
    constructor(props) {
        super(props);

        this.state = {
            toggle: true
        };

        // This is to bind context when passing onClick as a callback
        this.onClick = this.onClick.bind(this);
    }

    onClick() {
        this.setState((prevState, props) => ({
            toggle: !prevState.toggle
        }));
    }

    render() {
        return (
            <div onClick={this.onClick}>
                Hello, {this.props.name}! I am a SecondComponent.
                <br />
                Toggle is: {this.state.toggle}
            </div>
        );
    }
}
```

Durch das Erweitern einer Komponente mit `PureComponent` anstelle von `Component` wird die Lebenszyklusmethode " `shouldComponentUpdate()` " mit einem Vergleich der `shouldComponentUpdate()` flache Requisiten und `shouldComponentUpdate()` automatisch implementiert. Dadurch bleibt Ihre Anwendung leistungsfähiger, da weniger unnötige Renderings ausgeführt werden. Dies setzt voraus, dass Ihre Komponenten "Pure" sind und immer dieselbe Ausgabe mit demselben Status und Requisiteingang wiedergeben.

Komponenten höherer Ordnung

Komponenten höherer Ordnung (HOC) ermöglichen die gemeinsame Nutzung der Komponentenfunktionalität.

```
import React, { Component } from 'react';

const PrintHello = ComposedComponent => class extends Component {
    onClick() {
        console.log('hello');
    }
```

```
    /* The higher order component takes another component as a parameter
    and then renders it with additional props */
    render() {
        return <ComposedComponent {...this.props } onClick={this.onClick} />
    }
}

const FirstComponent = props => (
    <div onClick={ props.onClick }>
        Hello, {props.name}! I am a FirstComponent.
    </div>
);

const ExtendedComponent = PrintHello(FirstComponent);
```

Komponenten höherer Ordnung werden verwendet, wenn Sie die Logik für mehrere Komponenten gemeinsam nutzen möchten, unabhängig davon, wie unterschiedlich sie dargestellt werden.

setState-Fallstricke

Vorsicht ist geboten, wenn Sie `setState` in einem asynchronen Kontext verwenden. Sie könnten beispielsweise versuchen, `setState` im Callback einer Get-Anforderung `setState` :

```
class MyClass extends React.Component {
    constructor() {
        super();

        this.state = {
            user: {}
        };
    }

    componentDidMount() {
        this.fetchUser();
    }

    fetchUser() {
        $.get('/api/users/self')
            .then((user) => {
                this.setState({user: user});
            });
    }

    render() {
        return <h1>{this.state.user}</h1>;
    }
}
```

Dies kann zu Problemen führen. Wenn der Rückruf nach dem `this.setState` der `Component` `this.setState` wird, ist `this.setState` keine Funktion. Wenn dies der Fall ist, sollten Sie darauf achten, dass `setState` .

In diesem Beispiel möchten Sie möglicherweise die XHR-Anforderung abbrechen, wenn die Komponente abgemeldet wird:

```
class MyClass extends React.Component {
    constructor() {
        super();

        this.state = {
            user: {},
            xhr: null
        };
    }

    componentWillUnmount() {
        let xhr = this.state.xhr;

        // Cancel the xhr request, so the callback is never called
        if (xhr && xhr.readyState != 4) {
            xhr.abort();
        }
    }

    componentDidMount() {
        this.fetchUser();
    }

    fetchUser() {
        let xhr = $.get('/api/users/self')
            .then((user) => {
                this.setState({user: user});
            });

        this.setState({xhr: xhr});
    }
}
```

Die async-Methode wird als Status gespeichert. In `componentWillUnmount` Sie alle Bereinigungen durch, einschließlich des Abbruchs der XHR-Anforderung.

Sie könnten auch etwas komplexeres machen. In diesem Beispiel erstelle ich eine 'stateSetter'-Funktion, die das this-Objekt als Argument akzeptiert und `this.setState` verhindert, wenn die Funktion `cancel` aufgerufen wurde:

```
function stateSetter(context) {
    var cancelled = false;
    return {
        cancel: function () {
            cancelled = true;
        },
        setState(newState) {
            if (!cancelled) {
                context.setState(newState);
            }
        }
    }
}

class Component extends React.Component {
    constructor(props) {
        super(props);
        this.setter = stateSetter(this);
        this.state = {
```

```
            user: 'loading'
        };
    }
    componentWillUnmount() {
        this.setter.cancel();
    }
    componentDidMount() {
        this.fetchUser();
    }
    fetchUser() {
        $.get('/api/users/self')
            .then((user) => {
                this.setter.setState({user: user});
            });
    }
    render() {
        return <h1>{this.state.user}</h1>
    }
}
```

Dies funktioniert, weil die `cancelled` Variable in dem `setState` uns erstellten `setState` Abschluss sichtbar ist.

Requisiten

Requisiten sind eine Möglichkeit, Informationen an eine React-Komponente zu übergeben. Sie können einen beliebigen Typ haben, einschließlich Funktionen - manchmal auch als Rückrufe bezeichnet.

In JSX werden Requisiten mit der Attributsyntax übergeben

```
<MyComponent userID={123} />
```

Innerhalb der Definition für MyComponent kann jetzt auf das Benutzerobjekt zugegriffen werden

```
// The render function inside MyComponent
render() {
    return (
        <span>The user's ID is {this.props.userID}</span>
    )
}
```

Es ist wichtig, alle `props` , ihre Typen und gegebenenfalls ihren Standardwert zu definieren:

```
// defined at the bottom of MyComponent
MyComponent.propTypes = {
    someObject: React.PropTypes.object,
    userID: React.PropTypes.number.isRequired,
    title: React.PropTypes.string
};

MyComponent.defaultProps = {
    someObject: {},
    title: 'My Default Title'
}
```

In diesem Beispiel ist die `someObject` optional, aber die `userID` der `userID` ist erforderlich. Wenn Sie `userID` keine `userID` `MyComponent` , zeigt das React- `userID` zur Laufzeit eine Konsole an, die `userID` darauf `MyComponent` , dass die erforderliche Eigenschaft nicht bereitgestellt wurde. Achtung, diese Warnung wird nur in der Entwicklungsversion der React-Bibliothek angezeigt. Die Produktionsversion protokolliert keine Warnungen.

Die Verwendung von `defaultProps` ermöglicht Ihnen die Vereinfachung

```
const { title = 'My Default Title' } = this.props;
console.log(title);
```

zu

```
console.log(this.props.title);
```

Es ist auch ein Schutz für die Verwendung von `object` - `functions` `array` und `functions` . Wenn Sie keine Standard-Requisite für ein Objekt angeben, wird Folgendes ausgegeben, wenn die Requisite nicht übergeben wird:

```
if (this.props.someObject.someKey)
```

In obigem Beispiel ist `this.props.someObject` `undefined` und daher wird bei der Prüfung von `someKey` ein Fehler `someKey` , und der Code wird `someKey` . Mit `defaultProps` Sie die oben genannte Prüfung sicher verwenden.

Komponentenzustände - Dynamische Benutzeroberfläche

Angenommen, wir möchten das folgende Verhalten haben - Wir haben eine Überschrift (zum Beispiel ein h3-Element), und wenn Sie darauf klicken, möchten wir, dass es ein Eingabefeld wird, damit wir den Namen der Überschrift ändern können. React macht dies sehr einfach und intuitiv, indem Komponentenzustände und if-else-Anweisungen verwendet werden. (Code-Erklärung unten)

```
// I have used ReactBootstrap elements. But the code works with regular html elements also
var Button = ReactBootstrap.Button;
var Form = ReactBootstrap.Form;
var FormGroup = ReactBootstrap.FormGroup;
var FormControl = ReactBootstrap.FormControl;

var Comment = reactCreateClass({
  getInitialState: function() {
    return {show: false, newTitle: ''};
  },

  handleTitleSubmit: function() {
     //code to handle input box submit - for example, issue an ajax request to change name in
database
  },

  handleTitleChange: function(e) {
    //code to change the name in form input box. newTitle is initialized as empty string. We
```

```
need to update it with the string currently entered by user in the form
    this.setState({newTitle: e.target.value});
  },

  changeComponent: function() {
    // this toggles the show variable which is used  for dynamic UI
    this.setState({show: !this.state.show)};
  },

  render: function() {

    var clickableTitle;

    if(this.state.show) {
        clickableTitle = <Form inline onSubmit={this.handleTitleSubmit}>
                            <FormGroup controlId="formInlineTitle">
                                <FormControl type="text" onChange={this.handleTitleChange}>
                            </FormGroup>
                        </Form>;
    } else {
        clickabletitle = <div>
                           <Button bsStyle="link" onClick={this.changeComponent}>
                               <h3> Default Text </h3>
                           </Button>
                        </div>;
    }

    return (
        <div className="comment">
            {clickableTitle}
        </div>
    );
  }
});

ReactDOM.render(
    <Comment />, document.getElementById('content')
);
```

Der Hauptteil des Codes ist die Variable **clickableTitle** . Basierend auf der Zustandsvariablen **show** kann es sich entweder um ein Formularelement oder um ein Button-Element handeln. Reakt ermöglicht das Verschachteln von Komponenten.

Wir können also ein {clickableTitle} -Element in die Renderfunktion einfügen. Es sucht nach der Variable clickableTitle. Basierend auf dem Wert 'this.state.show' wird das entsprechende Element angezeigt.

Variationen zustandsloser Funktionskomponenten

```
const languages = [
  'JavaScript',
  'Python',
  'Java',
  'Elm',
  'TypeScript',
  'C#',
  'F#'
```

```
]
```

```
// one liner
const Language = ({language}) => <li>{language}</li>

Language.propTypes = {
  message: React.PropTypes.string.isRequired
}
```

```
/**
* If there are more than one line.
* Please notice that round brackets are optional here,
* However it's better to use them for readability
*/
const LanguagesList = ({languages}) => {
  <ul>
    {languages.map(language => <Language language={language} />)}
  </ul>
}

LanguagesList.PropTypes = {
  languages: React.PropTypes.array.isRequired
}
```

```
/**
 * This syntax is used if there are more work beside just JSX presentation
 * For instance some data manipulations needs to be done.
 * Please notice that round brackets after return are required,
 * Otherwise return will return nothing (undefined)
 */
const LanguageSection = ({header, languages}) => {
  // do some work
  const formattedLanguages = languages.map(language => language.toUpperCase())
  return (
    <fieldset>
      <legend>{header}</legend>
      <LanguagesList languages={formattedLanguages} />
    </fieldset>
  )
}

LanguageSection.PropTypes = {
  header: React.PropTypes.string.isRequired,
  languages: React.PropTypes.array.isRequired
}
```

```
ReactDOM.render(
  <LanguageSection
    header="Languages"
    languages={languages} />,
  document.getElementById('app')
)
```

Hier finden Sie ein Arbeitsbeispiel davon.

Komponenten online lesen: https://riptutorial.com/de/reactjs/topic/1185/komponenten

Kapitel 14: Komponenten höherer Ordnung

Einführung

Komponenten höherer Ordnung (kurz "HOC") ist ein Designmuster für reaktive Anwendungen, mit dem Komponenten mit wiederverwendbarem Code verbessert werden. Sie ermöglichen das Hinzufügen von Funktionen und Verhalten zu vorhandenen Komponentenklassen.

Ein HOC ist eine reine Javascript-Funktion, die eine Komponente als Argument akzeptiert und eine neue Komponente mit der erweiterten Funktionalität zurückgibt.

Bemerkungen

HOCs werden häufig in Bibliotheken von Drittanbietern verwendet. Wie die Redux Connect-Funktion.

Examples

Einfache Komponente höherer Ordnung

Angenommen, wir möchten jedes Mal, wenn die Komponente aktiviert wird, console.log verwenden:

hocLogger.js

```
export default function hocLogger(Component) {
  return class extends React.Component {
    componentDidMount() {
      console.log('Hey, we are mounted!');
    }
    render() {
      return <Component {...this.props} />;
    }
  }
}
```

Verwenden Sie dieses HOC in Ihrem Code:

MyLoggedComponent.js

```
import React from "react";
import {hocLogger} from "./hocLogger";

export class MyLoggedComponent extends React.Component {
    render() {
        return (
            <div>
                This component get's logged to console on each mount.
            </div>
```

```
        );
    }
}

// Now wrap MyLoggedComponent with the hocLogger function
export default hocLogger(MyLoggedComponent);
```

Komponente höherer Ordnung, die die Authentifizierung überprüft

Nehmen wir an, wir haben eine Komponente, die nur angezeigt werden soll, wenn der Benutzer angemeldet ist.

Also erstellen wir ein HOC, das die Authentifizierung für jedes render () prüft:

AuthenticatedComponent.js

```
import React from "react";

export function requireAuthentication(Component) {
    return class AuthenticatedComponent extends React.Component {

        /**
         * Check if the user is authenticated, this.props.isAuthenticated
         * has to be set from your application logic (or use react-redux to retrieve it from
global state).
         */
        isAuthenticated() {
            return this.props.isAuthenticated;
        }

        /**
         * Render
         */
        render() {
            const loginErrorMessage = (
                <div>
                    Please <a href="/login">login</a> in order to view this part of the
application.
                </div>
            );

            return (
                <div>
                    { this.isAuthenticated === true ? <Component {...this.props} /> :
loginErrorMessage }
                </div>
            );
        }
    };
}

export default requireAuthentication;
```

Wir verwenden dann diese Komponente höherer Ordnung in unseren Komponenten, die für anonyme Benutzer ausgeblendet werden sollten:

MyPrivateComponent.js

```
import React from "react";
import {requireAuthentication} from "./AuthenticatedComponent";

export class MyPrivateComponent extends React.Component {
    /**
     * Render
     */
    render() {
        return (
            <div>
                My secret search, that is only viewable by authenticated users.
            </div>
        );
    }
}

// Now wrap MyPrivateComponent with the requireAuthentication function
export default requireAuthentication(MyPrivateComponent);
```

Dieses Beispiel wird hier genauer beschrieben.

Komponenten höherer Ordnung online lesen:
https://riptutorial.com/de/reactjs/topic/9819/komponenten-hoherer-ordnung

Kapitel 15: Lebenszyklus der Reaktionskomponente

Einführung

Mithilfe von Lebenszyklusmethoden können Sie Code ausführen und mit Ihrer Komponente zu verschiedenen Zeitpunkten in der Komponentenlebensdauer interagieren. Diese Methoden basieren auf einer Komponente "Mounten", "Aktualisieren" und "Unmounting".

Examples

Komponentenerstellung

Wenn eine React-Komponente erstellt wird, werden einige Funktionen aufgerufen:

- Wenn Sie `React.createClass` (ES5) verwenden, werden 5 benutzerdefinierte Funktionen aufgerufen
- Wenn Sie die `class Component extends React.Component` (ES6) verwenden, werden 3 benutzerdefinierte Funktionen aufgerufen

`getDefaultProps()` (nur ES5)

Dies ist die **erste** aufgerufene Methode.

Von dieser Funktion zurückgegebene Prop-Werte werden als Standardwerte verwendet, wenn sie nicht definiert werden, wenn die Komponente instanziiert wird.

Im folgenden Beispiel wird `this.props.name` auf `Bob` sofern nicht anders angegeben:

```
getDefaultProps() {
  return {
    initialCount: 0,
    name: 'Bob'
  };
}
```

`getInitialState()` (nur ES5)

Dies ist die **zweite** aufgerufene Methode.

Der Rückgabewert von `getInitialState()` definiert den Anfangsstatus der React-Komponente. Das React-Framework ruft diese Funktion auf und weist diesem `this.state` den Rückgabewert zu.

Im folgende Beispiel `this.state.count` wird mit dem Wert von intialized `this.props.initialCount` :

```
getInitialState() {
  return {
    count : this.props.initialCount
  };
}
```

`componentWillMount()` (ES5 und ES6)

Dies ist die **dritte** aufgerufene Methode.

Diese Funktion kann verwendet werden, um endgültige Änderungen an der Komponente vorzunehmen, bevor sie zum DOM hinzugefügt wird.

```
componentWillMount() {
  ...
}
```

`render()` (ES5 und ES6)

Dies ist die **vierte** aufgerufene Methode.

Die Funktion `render()` sollte eine reine Funktion des Zustands und der Requisiten der Komponente sein. Sie gibt ein einzelnes Element zurück, das die Komponente während des Rendervorgangs darstellt, und sollte entweder eine native DOM-Komponente (z. B. `<p />`) oder eine Verbundkomponente sein. Wenn nichts gerendert werden soll, kann es `null` oder `undefined` .

Diese Funktion wird nach jeder Änderung der Requisiten oder des Zustands der Komponente aufgerufen.

```
render() {
  return (
    <div>
      Hello, {this.props.name}!
    </div>
  );
}
```

`componentDidMount()` (ES5 und ES6)

Dies ist die **fünfte** Methode, die aufgerufen wird.

Die Komponente wurde gemountet und Sie können jetzt auf die DOM-Knoten der Komponente zugreifen, z. B. über `refs` .

Diese Methode sollte verwendet werden für:

- Timer vorbereiten
- Daten abrufen
- Ereignis-Listener hinzufügen
- DOM-Elemente bearbeiten

```
componentDidMount() {
  ...
}
```

ES6-Syntax

Wenn die Komponente mit der ES6-Klassensyntax definiert ist, können die Funktionen `getDefaultProps()` und `getInitialState()` nicht verwendet werden.

Stattdessen deklarieren wir unsere `defaultProps` als statische Eigenschaft für die Klasse und deklarieren die Zustandsform und den Anfangszustand im Konstruktor unserer Klasse. Beide werden zur Konstruktionszeit auf die Instanz der Klasse festgelegt, bevor eine andere React-Lebenszyklusfunktion aufgerufen wird.

Das folgende Beispiel veranschaulicht diesen alternativen Ansatz:

```
class MyReactClass extends React.Component {
  constructor(props){
    super(props);

    this.state = {
      count: this.props.initialCount
    };
  }

  upCount() {
    this.setState((prevState) => ({
      count: prevState.count + 1
    }));
  }

  render() {
    return (
      <div>
        Hello, {this.props.name}!<br />
        You clicked the button {this.state.count} times.<br />
        <button onClick={this.upCount}>Click here!</button>
      </div>
    );
  }
}

MyReactClass.defaultProps = {
  name: 'Bob',
  initialCount: 0
};
```

`getDefaultProps()` ersetzen

Standardwerte für die Komponentenrequisiten werden durch Festlegen der Eigenschaft `defaultProps` der Klasse angegeben:

```
MyReactClass.defaultProps = {
  name: 'Bob',
  initialCount: 0
};
```

`getInitialState()` ersetzen

Der idiomatische Weg zum Einrichten des Anfangszustands der Komponente besteht darin, im Konstruktor `this.state` :

```
constructor(props){
  super(props);

  this.state = {
    count: this.props.initialCount
  };
}
```

Komponentenaktualisierung

`componentWillReceiveProps(nextProps)`

Dies ist die **erste Funktion, die bei Eigenschaftsänderungen aufgerufen wird** .

Wenn sich **die Eigenschaften der Komponente ändern** , ruft React diese Funktion mit den **neuen Eigenschaften auf** . Sie können mit *this.props* zu den alten Requisiten zuzugreifen und diese auf die neuen Requisiten mit *nextProps.*

Mit diesen Variablen können Sie einige Vergleichsoperationen zwischen alten und neuen Requisiten durchführen oder eine Funktion aufrufen, da sich eine Eigenschaft ändert.

```
componentWillReceiveProps(nextProps){
  if (nextProps.initialCount && nextProps.initialCount > this.state.count){
    this.setState({
      count : nextProps.initialCount
    });
  }
}
```

`shouldComponentUpdate(nextProps, nextState)`

Dies ist die **zweite Funktion, die bei Eigenschaftsänderungen und der ersten Zustandsänderung aufgerufen wird** .

Wenn eine andere Komponente / Ihre Komponente eine Eigenschaft / einen Status Ihrer Komponente ändert, **rendert React** standardmäßig eine neue Version Ihrer Komponente. In diesem Fall gibt diese Funktion immer true zurück.

Sie können diese Funktion überschreiben und **genauer wählen, ob Ihre Komponente aktualisiert werden muss oder nicht** .

Diese Funktion wird hauptsächlich zur **Optimierung verwendet** .

Wenn die Funktion " **false"** zurückgibt, wird die **Update-Pipeline sofort angehalten** .

```
componentShouldUpdate(nextProps, nextState){
  return this.props.name !== nextProps.name ||
    this.state.count !== nextState.count;
}
```

---

componentWillUpdate(nextProps, nextState)

Diese Funktion funktioniert wie `componentWillMount()` . **Änderungen sind nicht in DOM enthalten** . Sie können also einige Änderungen vornehmen, bevor das Update ausgeführt wird.

/! \: Sie können **this.setState ()** nicht verwenden.

```
componentWillUpdate(nextProps, nextState){}
```

---

render()

Es gibt einige Änderungen, also rendern Sie die Komponente erneut.

---

componentDidUpdate(prevProps, prevState)

Dasselbe wie `componentDidMount()` : **DOM wird aktualisiert** , sodass Sie hier einige Arbeiten am DOM ausführen können.

```
componentDidUpdate(prevProps, prevState){}
```

Komponentenentfernung

---

componentWillUnmount()

Diese Methode wird aufgerufen, **bevor** eine Komponente vom DOM abgehängt wird.

Es ist ein guter Ort für Reinigungsarbeiten wie:

- Ereignis-Listener entfernen
- Clearing-Timer
- Steckdosen stoppen
- Redux-Zustände bereinigen.

```
componentWillUnmount(){
  ...
}
```

Ein Beispiel zum Entfernen eines angehängten Ereignislisteners in `componentWillUnMount`

```
import React, { Component } from 'react';

export default class SideMenu extends Component {

  constructor(props) {
    super(props);
    this.state = {
        ...
      };
    this.openMenu = this.openMenu.bind(this);
    this.closeMenu = this.closeMenu.bind(this);
  }

  componentDidMount() {
    document.addEventListener("click", this.closeMenu);
  }

  componentWillUnmount() {
    document.removeEventListener("click", this.closeMenu);
  }

  openMenu() {
    ...
  }

  closeMenu() {
    ...
  }

  render() {
    return (
      <div>
        <a
          href      = "javascript:void(0)"
          className = "closebtn"
          onClick   = {this.closeMenu}
        >
          x
        </a>
        <div>
          Some other structure
        </div>
      </div>
    );
  }
}
```

Komponentenbehälter reagieren

Beim Erstellen einer React-Anwendung ist es häufig wünschenswert, Komponenten nach ihrer Hauptverantwortung in Präsentations- und Container-Komponenten zu unterteilen.
Bei Präsentationskomponenten handelt es sich nur um die Anzeige von Daten. Sie können als

Funktionen betrachtet werden und werden häufig als Funktionen implementiert, die ein Modell in eine Ansicht konvertieren. Normalerweise behalten sie keinen internen Zustand bei. Container-Komponenten befassen sich mit der Datenverwaltung. Dies kann intern durch den eigenen Bundesstaat oder durch Vermittlung bei einer Staatsverwaltungsbibliothek wie Redux erfolgen. Die Containerkomponente zeigt keine Daten direkt an, sondern leitet die Daten an eine Präsentationskomponente weiter.

```
// Container component
import React, { Component } from 'react';
import Api from 'path/to/api';

class CommentsListContainer extends Component {
    constructor() {
        super();
        // Set initial state
        this.state = { comments: [] }
    }

    componentDidMount() {
        // Make API call and update state with returned comments
        Api.getComments().then(comments => this.setState({ comments }));
    }

    render() {
        // Pass our state comments to the presentational component
        return (
            <CommentsList comments={this.state.comments} />;
        );
    }
}

// Presentational Component
const CommentsList = ({ comments }) => (
    <div>
        {comments.map(comment => (
            <div>{comment}</div>
        )}
    </div>
);

CommentsList.propTypes = {
    comments: React.PropTypes.arrayOf(React.PropTypes.string)
}
```

Lebenszyklus-Methodenaufruf in verschiedenen Status

Dieses Beispiel dient als Ergänzung zu anderen Beispielen, in denen die Verwendung der Lebenszyklusmethoden beschrieben wird und wann die Methode aufgerufen wird.

In diesem Beispiel wird zusammengefasst, welche Methoden (ComponentWillMount, ComponentWillReceiveProps usw.) aufgerufen werden und in welcher Reihenfolge sich die Komponenten **in verschiedenen Status unterscheiden** :

Wenn eine Komponente initialisiert wird:

1. getDefaultProps

2. getInitialState
3. KomponenteWillMount
4. machen
5. componentDidMount

Wenn sich eine Komponente geändert hat:

1. shouldComponentUpdate
2. KomponenteWillUpdate
3. machen
4. componentDidUpdate

Wenn eine Komponente Requisiten geändert hat:

1. componentWillReceiveProps
2. shouldComponentUpdate
3. KomponenteWillUpdate
4. machen
5. componentDidUpdate

Wenn eine Komponente abgemeldet wird:

1. KomponenteWillUnmount

Lebenszyklus der Reaktionskomponente online lesen:
https://riptutorial.com/de/reactjs/topic/2750/lebenszyklus-der-reaktionskomponente

Kapitel 16: Lösungen für die Benutzeroberfläche

Einführung

Nehmen wir an, wir lassen uns von einigen Ideen moderner Benutzeroberflächen inspirieren, die in Programmen verwendet werden, und konvertieren sie in React-Komponenten. Daraus besteht das Thema " **User Interface Solutions** ". Die Namensnennung ist genehmigt.

Examples

Grundbereich

```
import React from 'react';

class Pane extends React.Component {
    constructor(props) {
        super(props);
    }

    render() {
        return React.createElement(
            'section', this.props
        );
    }
}
```

Panel

```
import React from 'react';

class Panel extends React.Component {
    constructor(props) {
        super(props);
    }

    render(...elements) {
        var props = Object.assign({
            className: this.props.active ? 'active' : '',
            tabIndex: -1
        }, this.props);

        var css = this.css();
        if (css != '') {
            elements.unshift(React.createElement(
                'style', null,
                css
            ));
        }

        return React.createElement(
```

```
            'div', props,
            ...elements
        );
    }

    static title() {
        return '';
    }
    static css() {
        return '';
    }
}
```

Hauptunterschiede gegenüber dem einfachen Bereich sind:

- Das Panel hat den Fokus in der Instanz, wenn es per Skript aufgerufen oder mit der Maus angeklickt wird.
- Die statische Methode für jede Komponente des `title` verfügt über eine statische Methode. Daher kann sie um eine andere Panel-Komponente mit überschriebenem `title` (Grund dafür ist, dass die Funktion beim Rendern zu Lokalisierungszwecken erneut aufgerufen werden kann. In diesem Beispiel macht der `title` jedoch keinen Sinn.) ;
- Es kann einzelne Stylesheets enthalten, die in der statischen `css` Methode deklariert wurden (Sie können den Dateiinhalt aus `PANEL.css`).

Tab

```
import React from 'react';

class Tab extends React.Component {
    constructor(props) {
        super(props);
    }

    render() {
        var props = Object.assign({
            className: this.props.active ? 'active' : ''
        }, this.props);
        return React.createElement(
            'li', props,
            React.createElement(
                'span', props,
                props.panelClass.title()
            )
        );
    }
}
```

`panelClass` Eigenschaft `panelClass` der `Tab` Instanz muss die zur Beschreibung verwendete *Panel*-Klasse enthalten.

PanelGroup

```
import React from 'react';
import Tab from './Tab.js';
```

```
class PanelGroup extends React.Component {
    constructor(props) {
        super(props);
        this.setState({
            panels: props.panels
        });
    }

    render() {
        this.tabSet = [];
        this.panelSet = [];
        for (let panelData of this.state.panels) {
            var tabIsActive = this.state.activeTab == panelData.name;
            this.tabSet.push(React.createElement(
                Tab, {
                    name: panelData.name,
                    active: tabIsActive,
                    panelClass: panelData.class,
                    onMouseDown: () => this.openTab(panelData.name)
                }
            ));
            this.panelSet.push(React.createElement(
                panelData.class, {
                    id: panelData.name,
                    active: tabIsActive,
                    ref: tabIsActive ? 'activePanel' : null
                }
            ));
        }
        return React.createElement(
            'div', { className: 'PanelGroup' },
            React.createElement(
                'nav', null,
                React.createElement(
                    'ul', null,
                    ...this.tabSet
                )
            ),
            ...this.panelSet
        );
    }

    openTab(name) {
        this.setState({ activeTab: name });
        this.findDOMNode(this.refs.activePanel).focus();
    }
}
```

`panels` Eigenschaft der `PanelGroup` Instanz muss ein Array mit Objekten enthalten. Jedes Objekt dort erklärt wichtige Daten zu Panels:

- **`name`** - Kennung des Panels, das vom Controller-Skript verwendet wird;
- **`class`** - Klasse der Klasse.

Vergessen Sie nicht, die Eigenschaft `activeTab` auf den Namen der benötigten Registerkarte `activeTab` .

Klärung

Wenn die Registerkarte deaktiviert ist, erhält der benötigte Bereich den Klassennamen `active` DOM-Element (bedeutet, dass es sichtbar sein wird), und der Fokus ist jetzt aktiv.

Beispielansicht mit `PanelGroup`s

```
import React from 'react';
import Pane from './components/Pane.js';
import Panel from './components/Panel.js';
import PanelGroup from './components/PanelGroup.js';

class MainView extends React.Component {
    constructor(props) {
        super(props);
    }

    render() {
        return React.createElement(
            'main', null,
            React.createElement(
                Pane, { id: 'common' },
                React.createElement(
                    PanelGroup, {
                        panels: [
                            {
                                name: 'console',
                                panelClass: ConsolePanel
                            },
                            {
                                name: 'figures',
                                panelClass: FiguresPanel
                            }
                        ],
                        activeTab: 'console'
                    }
                )
            ),
            React.createElement(
                Pane, { id: 'side' },
                React.createElement(
                    PanelGroup, {
                        panels: [
                            {
                                name: 'properties',
                                panelClass: PropertiesPanel
                            }
                        ],
                        activeTab: 'properties'
                    }
                )
            )
        );
    }
}

class ConsolePanel extends Panel {
    constructor(props) {
```

```
        super(props);
    }

    static title() {
        return 'Console';
    }
}

class FiguresPanel extends Panel {
    constructor(props) {
        super(props);
    }

    static title() {
        return 'Figures';
    }
}

class PropertiesPanel extends Panel {
    constructor(props) {
        super(props);
    }

    static title() {
        return 'Properties';
    }
}
```

Lösungen für die Benutzeroberfläche online lesen:
https://riptutorial.com/de/reactjs/topic/8112/losungen-fur-die-benutzeroberflache

Kapitel 17: Performance

Examples

Die Grundlagen - HTML DOM vs. Virtual DOM

HTML-DOM ist teuer

Jede Webseite wird intern als Objektbaum dargestellt. Diese Darstellung wird als *Document Object Model bezeichnet* . Darüber hinaus handelt es sich um eine sprachneutrale Schnittstelle, über die Programmiersprachen (wie JavaScript) auf die HTML-Elemente zugreifen können.

Mit anderen Worten

> Das HTML-DOM ist ein Standard zum Abrufen, Ändern, Hinzufügen oder Löschen von HTML-Elementen.

Diese **DOM-Vorgänge** sind jedoch extrem **teuer** .

Virtual DOM ist eine Lösung

Das Team von React kam daher auf die Idee, das *HTML-DOM* zu abstrahieren und ein eigenes *virtuelles DOM zu* erstellen, um die Mindestanzahl an Operationen zu berechnen, die auf dem *HTML-DOM ausgeführt werden* müssen, um den aktuellen Status unserer Anwendung zu replizieren.

Das virtuelle DOM spart Zeit durch unnötige DOM-Änderungen.

Wie genau?

Zu jedem Zeitpunkt wird der Anwendungsstatus von React als `Virtual DOM` . Bei jeder Änderung des Anwendungsstatus werden diese Schritte von React ausgeführt, um die Leistung zu optimieren

1. Generieren Sie ein neues *virtuelles DOM* , das den neuen Status unserer Anwendung darstellt
2. Vergleichen Sie das alte virtuelle DOM (das das aktuelle HTML-DOM darstellt) mit dem neuen virtuellen DOM
3. Basierend auf 2. ermitteln Sie die Mindestanzahl an Operationen, um das alte virtuelle DOM (das das aktuelle HTML-DOM darstellt) in das neue virtuelle DOM umzuwandeln

- Um mehr darüber zu erfahren, lesen Sie den Diff-Algorithmus von React

4. Nachdem diese Vorgänge gefunden wurden, werden sie ihren entsprechenden *HTML-DOM-* Vorgängen zugeordnet

- Denken Sie daran, dass das *virtuelle DOM* nur eine Abstraktion des *HTML-DOM ist* und zwischen diesen eine isomorphe Beziehung besteht

5. Jetzt wird die minimale Anzahl von Operationen, die gefunden und in ihre entsprechenden *HTML-DOM-* Operationen übertragen wurden, nun direkt auf das *HTML-DOM* der Anwendung angewendet, wodurch Zeit eingespart wird, die das *HTML-DOM* unnötig *ändert* .

Hinweis: Auf das virtuelle DOM angewendete Vorgänge sind billig, da das virtuelle DOM ein JavaScript-Objekt ist.

Der Diff-Algorithmus von React

Die Erzeugung der Mindestanzahl von Operationen zum Umwandeln eines Baums in einen anderen hat eine Komplexität in der Reihenfolge O (n ^ 3), wobei n die Anzahl der Knoten in dem Baum ist. React stützt sich auf zwei Annahmen, um dieses Problem in einer linearen Zeit zu lösen - O (n)

1. Zwei Komponenten derselben Klasse erzeugen ähnliche Bäume, und zwei Komponenten verschiedener Klassen erzeugen unterschiedliche Bäume.

2. Es ist möglich, einen eindeutigen Schlüssel für Elemente bereitzustellen, der für verschiedene Renderings stabil ist.

Um zu entscheiden, ob zwei Knoten unterschiedlich sind, unterscheidet React drei Fälle

1. Zwei Knoten unterscheiden sich, wenn sie unterschiedliche Typen haben.

- Beispielsweise unterscheidet sich `<div>...</div>` von `<span>...</span>`

2. Immer wenn zwei Knoten unterschiedliche Schlüssel haben

- Beispielsweise unterscheidet sich `<div key="1">...</div>` von `<div key="2">...</div>`

Darüber hinaus ist **das Folgende entscheidend und äußerst wichtig,** wenn Sie die Leistung optimieren möchten

> Wenn sie [zwei Knoten] nicht vom selben Typ sind, wird React nicht einmal versuchen, das zu vergleichen, was sie rendern. Es wird nur die erste aus dem DOM entfernt und die zweite eingefügt.

Hier ist der Grund

> Es ist sehr unwahrscheinlich, dass ein Element ein DOM erzeugt, das wie ein Element aussehen würde. Anstatt zu versuchen, diese beiden Strukturen abzugleichen, baut React den Baum einfach von Grund auf neu auf.

Tipps

Wenn zwei Knoten nicht vom selben Typ sind, versucht React nicht, sie zuzuordnen - es entfernt lediglich den ersten Knoten aus dem DOM und fügt den zweiten ein. Deshalb sagt der erste Tipp

1. Wenn Sie sehen, dass Sie zwischen zwei Komponentenklassen mit sehr ähnlicher Ausgabe wechseln, möchten Sie sie möglicherweise derselben Klasse zuordnen.

---

2. Verwenden Sie shouldComponentUpdate, um zu verhindern, dass die Komponente erneut ausgegeben wird, wenn Sie wissen, dass sich die Komponente zum Beispiel nicht ändert

```
shouldComponentUpdate: function(nextProps, nextState) {
  return nextProps.id !== this.props.id;
}
```

Leistungsmessung mit ReactJS

Sie können nichts verbessern, was Sie nicht messen können . Um die Leistung von React-Komponenten zu verbessern, sollten Sie sie messen können. ReactJS bietet mit *angebauter* Tools Leistung zu messen. Importieren Sie das `react-addons-perf` Modul, um die Leistung zu messen

```
import Perf from 'react-addons-perf' // ES6
var Perf = require('react-addons-perf') // ES5 with npm
var Perf = React.addons.Perf; // ES5 with react-with-addons.js
```

Sie können die folgenden Methoden aus dem importierten `Perf` Modul verwenden:

- Perf.printInclusive ()
- Perf.printExclusive ()
- Perf.printWasted ()
- Perf.printOperations ()
- Perf.printDOM ()

Das wichtigste, das Sie meistens benötigen werden, ist `Perf.printWasted()` das Ihnen die tabellarische Darstellung der verschwendeten Zeit Ihrer einzelnen Komponente gibt

(index)	Owner > component	Waste
0	"Todos > TodoItem"	102.7

Total time: 132.71 ms

Sie können die Spalte " **Verschwendete Zeit"** in der Tabelle beachten und die Leistung der Komponente mithilfe des obigen **Tipps & Tricks** verbessern

Lesen Sie den offiziellen React-Leitfaden und den ausgezeichneten Artikel von Benchling Engg. auf React Performance

Performance online lesen: https://riptutorial.com/de/reactjs/topic/6875/performance

Kapitel 18: React with Flow verwenden

Einführung

Verwendung des Flows Type Checkers zum Überprüfen von Typen in React-Komponenten.

Bemerkungen

Durchfluss | Reagieren

Examples

Verwenden von Flow zum Prüfen von Statustypen von statuslosen Funktionskomponenten

```
type Props = {
  posts: Array<Article>,
  dispatch: Function,
  children: ReactElement
}

const AppContainer =
  ({ posts, dispatch, children }: Props) => (
    <div className="main-app">
      <Header {...{ posts, dispatch }} />
      {children}
    </div>
  )
```

Verwenden von Flow zum Überprüfen der Stempeltypen

```
import React, { Component } from 'react';

type Props = {
  posts: Array<Article>,
  dispatch: Function,
  children: ReactElement
}

class Posts extends Component {
  props: Props;

  render () {
    // rest of the code goes here
  }
}
```

React with Flow verwenden online lesen: https://riptutorial.com/de/reactjs/topic/7918/react-with-flow-verwenden

Kapitel 19: React.createClass vs erweitert React.Component

Syntax

- Fall 1: React.createClass ({})
- Fall 2: Klasse MyComponent erweitert React.Component {}

Bemerkungen

`React.createClass` wurde in Version 15.5 nicht mehr unterstützt und wird voraussichtlich in Version 16 entfernt . Für diejenigen, die es noch benötigen, gibt es ein Drop-In-Ersatzpaket . Beispiele, die es verwenden, sollten aktualisiert werden.

Examples

Reakt-Komponente erstellen

Untersuchen wir die Syntaxunterschiede durch Vergleich zweier Codebeispiele.

React.createClass (veraltet)

Hier haben wir ein `const` mit einer zugewiesenen React-Klasse, mit der `render` Funktion eine typische Definition der Basiskomponente abschließen.

```
import React from 'react';

const MyComponent = React.createClass({
  render() {
    return (
      <div></div>
    );
  }
});

export default MyComponent;
```

React.Component

Nehmen wir die obige Definition von React.createClass und konvertieren sie in eine ES6-Klasse.

```
import React from 'react';
```

```
class MyComponent extends React.Component {
  render() {
    return (
      <div></div>
    );
  }
}

export default MyComponent;
```

In diesem Beispiel verwenden wir jetzt ES6-Klassen. Für die React-Änderungen erstellen wir jetzt eine Klasse mit dem Namen **MyComponent** und erweitern sie von React.Component, anstatt direkt auf React.createClass zuzugreifen. Auf diese Weise verwenden wir weniger React Boilerplate und mehr JavaScript.

PS: Normalerweise wird dies mit etwas wie Babel verwendet, um die ES6 auf ES5 zu kompilieren, damit sie in anderen Browsern funktioniert.

Deklarieren Sie Standard-Requisiten und PropTypes

Es gibt wichtige Änderungen bei der Verwendung und Deklaration von Standard-Requisiten und deren Typen.

React.createClass

In dieser Version ist die Eigenschaft `propTypes` ein Objekt, in dem wir den Typ für jede Eigenschaft `propTypes` können. Die `getDefaultProps` Eigenschaft ist eine Funktion, die ein Objekt zurückgibt, um die ursprünglichen Requisiten zu erstellen.

```
import React from 'react';

const MyComponent = React.createClass({
  propTypes: {
    name: React.PropTypes.string,
    position: React.PropTypes.number
  },
  getDefaultProps() {
    return {
      name: 'Home',
      position: 1
    };
  },
  render() {
    return (
      <div></div>
    );
  }
});

export default MyComponent;
```

React.Component

Diese Version verwendet `propTypes` als Eigenschaft der tatsächlichen **MyComponent-** Klasse anstelle einer Eigenschaft als Teil des `createClass` Definitionsobjekts.

Das `getDefaultProps` hat sich jetzt zu einer Object-Eigenschaft in der Klasse namens defaultProps geändert, da es nicht länger eine "get" -Funktion ist, sondern nur ein Object. Es vermeidet mehr React Boilerplate, dies ist nur reines JavaScript.

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div></div>
    );
  }
}
MyComponent.propTypes = {
  name: React.PropTypes.string,
  position: React.PropTypes.number
};
MyComponent.defaultProps = {
  name: 'Home',
  position: 1
};

export default MyComponent;
```

Darüber hinaus gibt es eine andere Syntax für `propTypes` und `defaultProps` . Dies ist eine Verknüpfung, wenn in Ihrem Build die ES7-Eigenschaftsinitialisierer aktiviert sind:

```
import React from 'react';

class MyComponent extends React.Component {
  static propTypes = {
    name: React.PropTypes.string,
    position: React.PropTypes.number
  };
  static defaultProps = {
    name: 'Home',
    position: 1
  };
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div></div>
    );
  }
}
```

```
export default MyComponent;
```

Anfangszustand einstellen

Es gibt Änderungen in der Einstellung der Anfangszustände.

React.createClass

Wir haben eine `getInitialState` Funktion, die einfach ein Objekt mit Anfangszuständen zurückgibt.

```
import React from 'react';

const MyComponent = React.createClass({
  getInitialState () {
    return {
      activePage: 1
    };
  },
  render() {
    return (
      <div></div>
    );
  }
});

export default MyComponent;
```

React.Component

In dieser Version deklarieren wir all state als einfache **Initialisierungseigenschaft im Konstruktor** , anstatt die Funktion `getInitialState` verwenden. Es fühlt sich weniger nach "React API" an, da dies lediglich JavaScript ist.

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      activePage: 1
    };
  }
  render() {
    return (
      <div></div>
    );
  }
}

export default MyComponent;
```

Mixins

Wir können `mixins` nur mit der Methode React.createClass verwenden.

React.createClass

In dieser Version können Sie `mixins` zu Komponenten hinzufügen, indem Sie die mixins - Eigenschaft verwenden, die ein Array verfügbarer Mixins benötigt. Diese erweitern dann die Komponentenklasse.

```
import React from 'react';

var MyMixin = {
  doSomething() {

  }
};
const MyComponent = React.createClass({
  mixins: [MyMixin],
  handleClick() {
    this.doSomething(); // invoke mixin's method
  },
  render() {
    return (
      <button onClick={this.handleClick}>Do Something</button>
    );
  }
});

export default MyComponent;
```

React.Component

React Mixins werden nicht unterstützt, wenn in ES6 geschriebene React-Komponenten verwendet werden. Außerdem werden sie in React keine ES6-Klassen unterstützen. Der Grund ist, dass sie als schädlich betrachtet werden .

"dieser" Kontext

Die Verwendung von React.createClass bindet `this` Kontext (Werte) automatisch korrekt. `this` ist jedoch bei Verwendung von ES6-Klassen nicht der Fall.

React.createClass

Beachten Sie die `onClick` Deklaration mit der `this.handleClick` Methode. Wenn diese Methode aufgerufen wird, wendet React den richtigen Ausführungskontext auf den `handleClick` .

```
import React from 'react';

const MyComponent = React.createClass({
  handleClick() {
    console.log(this); // the React Component instance
  },
  render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
});

export default MyComponent;
```

React.Component

Bei ES6-Klassen ist `this` standardmäßig `null` . Die Eigenschaften der Klasse werden nicht automatisch an die React-Klasseninstanz (Komponente) gebunden.

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  handleClick() {
    console.log(this); // null
  }
  render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
}

export default MyComponent;
```

Es gibt einige Möglichkeiten, wie wir `this` Kontext richtig binden können.

Fall 1: Inline binden:

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  handleClick() {
    console.log(this); // the React Component instance
  }
  render() {
    return (
```

```
      <div onClick={this.handleClick.bind(this)}></div>
    );
  }
}

export default MyComponent;
```

Fall 2: Binden Sie im Klassenkonstruktor

Ein anderer Ansatz besteht darin, den Kontext von `this.handleClick` im `constructor` `this.handleClick` . Auf diese Weise vermeiden wir Inline-Wiederholungen. Von vielen als ein besserer Ansatz betrachtet, der JSX überhaupt nicht berührt:

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    console.log(this); // the React Component instance
  }
  render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
}

export default MyComponent;
```

Fall 3: Verwenden Sie die anonyme Funktion von ES6

Sie können auch eine anonyme ES6-Funktion verwenden, ohne explizit binden zu müssen:

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  handleClick = () => {
    console.log(this); // the React Component instance
  }
  render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
}

export default MyComponent;
```

ES6 / Reagieren Sie dieses Schlüsselwort mit ajax, um Daten vom Server abzurufen

```
import React from 'react';

class SearchEs6 extends React.Component{
    constructor(props) {
        super(props);
        this.state = {
            searchResults: []
        };
    }

    showResults(response){
        this.setState({
            searchResults: response.results
        })
    }

    search(url){
        $.ajax({
            type: "GET",
            dataType: 'jsonp',
            url: url,
            success: (data) => {
                this.showResults(data);
            },
            error: (xhr, status, err) => {
                console.error(url, status, err.toString());
            }
        });
    }

    render() {
        return (
            <div>
                <SearchBox search={this.search.bind(this)} />
                <Results searchResults={this.state.searchResults} />
            </div>
        );
    }
}
```

React.createClass vs erweitert React.Component online lesen:
https://riptutorial.com/de/reactjs/topic/6371/react-createclass-vs-erweitert-react-component

Kapitel 20: ReactJS auf Flux-Weise verwenden

Einführung

Es ist sehr praktisch, den Flux-Ansatz zu verwenden, wenn Ihre Anwendung mit ReactJS am Frontend aufgrund von begrenzten Strukturen und ein wenig neuem Code erweitert werden soll, um Statusänderungen in der Laufzeit zu erleichtern.

Bemerkungen

Flux ist die Anwendungsarchitektur, die Facebook verwendet, um clientseitige Webanwendungen zu erstellen. Es ergänzt die komponierbaren Ansichtskomponenten von React, indem ein unidirektionaler Datenfluss verwendet wird. Es ist eher ein Muster als ein formaler Rahmen, und Sie können Flux sofort ohne viel neuen Code verwenden.

Flussanwendungen bestehen aus drei Hauptteilen: *dem Dispatcher* , *den Filialen* und *den Ansichten* (React-Komponenten). Diese sollten nicht mit Model-View-Controller verwechselt werden. Controller sind zwar in einer Flux-Anwendung vorhanden, aber es handelt sich um Controller-Ansichten. Ansichten, die sich häufig oben in der Hierarchie befinden und Daten aus den Stores abrufen und an ihre untergeordneten Elemente weiterleiten. Außerdem werden Aktionsersteller - Dispatcher-Hilfsmethoden - zur Unterstützung einer semantischen API verwendet, die alle Änderungen beschreibt, die in der Anwendung möglich sind. Es kann hilfreich sein, sie als vierten Teil des Flux-Aktualisierungszyklus zu betrachten.

Flux verzichtet auf MVC auf einen unidirektionalen Datenfluss. Wenn ein Benutzer mit einer React-Ansicht interagiert, leitet die Ansicht eine Aktion über einen zentralen Dispatcher an die verschiedenen Stores weiter, die die Daten und die Geschäftslogik der Anwendung enthalten, wodurch alle betroffenen Views aktualisiert werden. Dies funktioniert besonders gut mit dem deklarativen Programmierstil von React, der es dem Geschäft ermöglicht, Aktualisierungen zu senden, ohne festzulegen, wie Ansichten zwischen Zuständen wechseln sollen.

Examples

Datenfluss

> Dies ist eine Übersicht der umfassenden Übersicht .

Das Flussmuster setzt die Verwendung eines unidirektionalen Datenflusses voraus.

1. **Aktion** - einfach Objekt beschreibt Aktion `type` und andere Eingabedaten.
2. **Dispatcher** - Single Action Receiver und Callbacks Controller. Stellen Sie sich vor, es ist ein zentraler Knotenpunkt Ihrer Anwendung.

3. **Store** - enthält den Anwendungsstatus und die Logik. Er registriert den Rückruf im Dispatcher und gibt ein Ereignis aus, um anzuzeigen, wann eine Änderung der Datenschicht stattgefunden hat.

4. **Ansicht** - Reagiert eine Komponente, die ein Änderungsereignis und Daten vom Speicher empfängt. Es bewirkt ein erneutes Rendern, wenn etwas geändert wird.

    > Ab dem Flux-Datenfluss können Ansichten auch **Aktionen erstellen** und diese für Benutzerinteraktionen an den Dispatcher übergeben.

Umgekehrt

Um es klarer zu machen, können wir am Ende beginnen.

- Verschiedene React-Komponenten (*Ansichten*) erhalten Daten aus verschiedenen Speichern über vorgenommene Änderungen.

    > Nur wenige Komponenten können **Controller-Views genannt werden** , da sie den Glue-Code bereitstellen, um die Daten aus den Filialen zu erhalten und Daten in der Kette ihrer Nachkommen weiterzugeben. Controller-Ansichten repräsentieren alle wichtigen Bereiche der Seite.

- *Speicher* können als Rückrufe bezeichnet werden, die den Aktionstyp und andere Eingabedaten für die Geschäftslogik Ihrer Anwendung vergleichen.

- *Der Dispatcher* ist ein allgemeiner Empfänger für Aktionen und ein Callback-Container.

- *Aktionen* sind nichts anderes als einfache Objekte mit dem erforderlichen `type` Eigenschaft.

    > Früher sollten Sie Konstanten für Aktionstypen und Hilfsmethoden (so genannte **Aktionsersteller**) verwenden.

ReactJS auf Flux-Weise verwenden online lesen:
https://riptutorial.com/de/reactjs/topic/8158/reactjs-auf-flux-weise-verwenden

Kapitel 21: ReactJS mit jQuery verwenden

Examples

ReactJS mit jQuery

Zunächst müssen Sie die Jquery-Bibliothek importieren. Wir müssen außerdem findDOmNode importieren, da wir den Dom bearbeiten werden. Und natürlich importieren wir auch React.

```
import React from 'react';
import { findDOMNode } from 'react-dom';
import $ from 'jquery';
```

Wir setzen eine Pfeilfunktion 'handleToggle', die ausgelöst wird, wenn ein Symbol angeklickt wird. Wir zeigen und verbergen nur ein Div mit einem Referenznamen 'toggle' onClick über ein Symbol.

```
handleToggle = () => {
    const el = findDOMNode(this.refs.toggle);
    $(el).slideToggle();
};
```

Setzen wir nun die Referenzbenennung 'toggle'

```
<ul className="profile-info additional-profile-info-list" ref="toggle">
  <li>
    <span className="info-email">Office Email</span>   me@shuvohabib.com
  </li>
</ul>
```

Das div-Element, bei dem der 'handleToggle' auf onClick ausgelöst wird.

```
 <div className="ellipsis-click" onClick={this.handleToggle}>
   <i className="fa-ellipsis-h"/>
 </div>
```

Lassen Sie sich den vollständigen Code unten ansehen, wie er aussieht.

```
import React from 'react';
import { findDOMNode } from 'react-dom';
import $ from 'jquery';

export default class FullDesc extends React.Component {
    constructor() {
        super();
    }

    handleToggle = () => {
        const el = findDOMNode(this.refs.toggle);
        $(el).slideToggle();
    };
```

```
    render() {
        return (
            <div className="long-desc">
                <ul className="profile-info">
                    <li>
                        <span className="info-title">User Name : </span> Shuvo Habib
                    </li>
                </ul>

                <ul className="profile-info additional-profile-info-list" ref="toggle">
                    <li>
                        <span className="info-email">Office Email</span> me@shuvohabib.com
                    </li>
                </ul>

                <div className="ellipsis-click" onClick={this.handleToggle}>
                    <i className="fa-ellipsis-h"/>
                </div>
            </div>
        );
    }
}
```

Wir sind fertig! Auf diese Weise können wir **jQuery in der React-** Komponente verwenden.

ReactJS mit jQuery verwenden online lesen: https://riptutorial.com/de/reactjs/topic/6009/reactjs-mit-jquery-verwenden

Kapitel 22: ReactJS mit Typescript verwenden

Examples

ReactJS-Komponente in Typescript geschrieben

Tatsächlich können Sie die ReactJS-Komponenten in Typescript wie in Facebook verwenden. Ersetzen Sie einfach die Dateierweiterung "jsx" in "tsx":

```
//helloMessage.tsx:
var HelloMessage = React.createClass({
  render: function() {
    return <div>Hello {this.props.name}</div>;
  }
});
ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

Um jedoch die Hauptfunktion von Typescript (statische Typprüfung) voll ausnutzen zu können, sollten einige Dinge ausgeführt werden:

1) Konvertieren Sie das React.createClass-Beispiel in die ES6-Klasse:

```
//helloMessage.tsx:
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}
ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

2) Als nächstes fügen Sie Props und Statusschnittstellen hinzu:

```
interface IHelloMessageProps {
    name:string;
}

interface IHelloMessageState {
  //empty in our case
}

class HelloMessage extends React.Component<IHelloMessageProps, IHelloMessageState> {
  constructor(){
    super();
  }
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}
ReactDOM.render(<HelloMessage name="Sebastian" />, mountNode);
```

Jetzt zeigt Typescript einen Fehler an, wenn der Programmierer vergisst, Requisiten zu übergeben. Oder wenn sie Requisiten hinzugefügt haben, die nicht in der Schnittstelle definiert sind.

Stateless React-Komponenten in Typescript

Reagieren Sie Komponenten, die reine Funktionen ihrer Requisiten sind und keinen internen Status erfordern, können Sie als JavaScript-Funktionen schreiben, anstatt die Standardklassensyntax zu verwenden:

```
import React from 'react'

const HelloWorld = (props) => (
    <h1>Hello, {props.name}!</h1>
);
```

Dasselbe kann in Typescript mit der `React.SFC` Klasse erreicht werden:

```
import * as React from 'react';

class GreeterProps {
   name: string
}

const Greeter : React.SFC<GreeterProps> = props =>
    <h1>Hello, {props.name}!</h1>;
```

Beachten Sie, dass der Name `React.SFC` ein Alias für `React.SFC` ist. `React.StatelessComponent` können verwendet werden.

Installation und Einrichtung

Um Typescript mit reagieren in einem Knotenprojekt verwenden zu können, müssen Sie zunächst ein Projektverzeichnis mit npm initialisieren. Um das Verzeichnis mit `npm init` zu initialisieren

Einbau über NPM oder Garn

Sie können React mit npm installieren, indem Sie folgende Schritte ausführen :

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

Facebook hat seinen eigenen Paketmanager namens Yarn veröffentlicht , mit dem auch React installiert werden kann. Nach der Installation von Yarn müssen Sie nur diesen Befehl ausführen:

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

Sie können React dann in Ihrem Projekt genauso verwenden, als wenn Sie React über npm installiert hätten.

Installation von Definitionstypen in Typescript 2.0+

Um Ihren Code mithilfe von Typoskript zu kompilieren, fügen Sie Typendefinitionsdateien mit npm oder gar ein.

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

oder unter Verwendung von Garn

```
yarn add --dev @types/react @types/react-dom
```

Installieren von Definitionstypen in älteren Versionen von Typescript

Sie müssen ein separates Paket mit dem Namen tsd verwenden

```
tsd install react react-dom --save
```

Hinzufügen oder Ändern der Typescript-Konfiguration

Um JSX , eine Sprache, die Javascript mit HTML / XML mischt, zu verwenden, müssen Sie die TypScript-Compiler-Konfiguration ändern. In der `tsconfig.json` Konfigurationsdatei des Projekts (normalerweise `tsconfig.json`) müssen Sie die JSX-Option hinzufügen:

```
"compilerOptions": {
    "jsx": "react"
},
```

Diese Compileroption weist den Typescript-Compiler grundsätzlich an, die JSX-Tags im Code in Javascript-Funktionsaufrufe zu übersetzen.

Um zu vermeiden, dass der TypScript-Compiler JSX in einfache JavaScript-Funktionsaufrufe konvertiert, verwenden Sie

```
"compilerOptions": {
    "jsx": "preserve"
},
```

Zustandslose und eigenschaftsfreie Komponenten

Die einfachste Reaktionskomponente ohne Zustand und ohne Eigenschaften kann wie folgt geschrieben werden:

```
import * as React from 'react';

const Greeter = () => <span>Hello, World!</span>
```

Diese Komponente kann jedoch nicht auf `this.props` zugreifen, da `this.props` nicht `this.props` kann, ob es sich um eine reagierende Komponente handelt. Um auf die Requisiten zuzugreifen, verwenden Sie:

```
import * as React from 'react';

const Greeter: React.SFC<{}> = props => () => <span>Hello, World!</span>
```

Selbst wenn die Komponente keine explizit definierten Eigenschaften hat, kann sie jetzt auf **`props.children`** zugreifen, da alle Komponenten von Natur aus **`props.children`** sind.

Eine andere ähnliche gute Verwendung von zustandslosen und eigenschaftslosen Komponenten ist die einfache Seitenvorlage. Im Folgenden ist eine examplinary einfache `Page` Komponente, vorausgesetzt , es gibt hypothetische `Container` , `NavTop` und `NavBottom` Komponenten bereits im Projekt:

```
import * as React from 'react';

const Page: React.SFC<{}> = props => () =>
    <Container>
        <NavTop />
        {props.children}
        <NavBottom />
    </Container>

const LoginPage: React.SFC<{}> = props => () =>
    <Page>
        Login Pass: <input type="password" />
    </Page>
```

In diesem Beispiel kann die `Page` Komponente später von jeder anderen tatsächlichen Seite als Basisvorlage verwendet werden.

ReactJS mit Typescript verwenden online lesen:
https://riptutorial.com/de/reactjs/topic/1419/reactjs-mit-typescript-verwenden

Kapitel 23: Reagiere mit Redux

Einführung

Redux ist heutzutage zum Status Quo für die Verwaltung des Status auf Anwendungsebene im Frontend geworden. In diesem Thema wird erläutert, warum und wie Sie die Statusverwaltungsbibliothek Redux in Ihren React-Anwendungen verwenden sollten.

Bemerkungen

Die komponentengesteuerte Architektur von React eignet sich hervorragend für die Aufgliederung der Anwendung in modulare, gekapselte kleine Teile, bringt jedoch einige Herausforderungen mit sich, um den Status der Anwendung als Ganzes zu verwalten. Die Zeit für die Verwendung von Redux ist, wenn Sie dieselben Daten über mehrere Komponenten oder Seiten (auch als Route bezeichnet) anzeigen möchten. Zu diesem Zeitpunkt können Sie die Daten nicht mehr in lokalen Variablen für die eine oder die andere Komponente speichern, und das Senden von Nachrichten zwischen Komponenten wird schnell zum Chaos. Mit Redux abonnieren Ihre Komponenten alle dieselben gemeinsam genutzten Daten im Geschäft, und der Zustand kann somit problemlos in der gesamten Anwendung konsistent dargestellt werden.

Examples

Verbinden verwenden

Erstellen Sie einen Redux-Store mit *createStore* .

```
import { createStore } from 'redux'
import todoApp from './reducers'
let store = createStore(todoApp, { inistialStateVariable: "derp"})
```

Verwenden Sie *connect* , um eine Komponente mit dem Redux-Store zu verbinden und Requisiten von Store zu Component zu ziehen.

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

const VisibleTodoList = connect(
  mapStateToProps,
  mapDispatchToProps
)(TodoList)

export default VisibleTodoList
```

Definieren Sie Aktionen, mit denen Ihre Komponenten Nachrichten an den Redux-Store senden können.

```
/*
```

```
 * action types
 */

export const ADD_TODO = 'ADD_TODO'

export function addTodo(text) {
  return { type: ADD_TODO, text }
}
```

Behandeln Sie diese Nachrichten und erstellen Sie einen neuen Status für das Speichern in Reduzierfunktionen.

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    default:
      return state
  }
}
```

Reagiere mit Redux online lesen: https://riptutorial.com/de/reactjs/topic/10856/reagiere-mit-redux

Kapitel 24: Reaktionsformen

Examples

Kontrollierte Komponenten

Eine kontrollierte Komponente ist an einen Wert gebunden, und ihre Änderungen werden mithilfe von ereignisbasierten Rückrufen im Code verarbeitet.

```
class CustomForm extends React.Component {
constructor() {
    super();
    this.state = {
        person: {
            firstName: '',
            lastName: ''
        }
    }
}

handleChange(event) {
    let person = this.state.person;
    person[event.target.name] = event.target.value;
    this.setState({person});
}

render() {
    return (
        <form>
            <input
            type="text"
            name="firstName"
            value={this.state.firstName}
            onChange={this.handleChange.bind(this)} />

            <input
            type="text"
            name="lastName"
            value={this.state.lastName}
            onChange={this.handleChange.bind(this)} />
        </form>
    )
}
```

}

In diesem Beispiel initialisieren wir state mit einem leeren Personenobjekt. Wir binden dann die Werte der 2 Eingänge an die einzelnen Tasten des Personenobjekts. `handleChange` der Benutzer den Typ `handleChange` , erfassen wir jeden Wert in der `handleChange` Funktion. Da die Werte der Komponenten festgeschrieben sind, können wir sie als die Benutzertypen durch Aufruf von `setState()` .

HINWEIS: Nicht Aufruf `setState()` , wenn sie mit gesteuerten Komponenten handeln, bewirkt ,

dass der Benutzer schreiben, aber nicht die Eingabe , weil Veränderungen reagieren macht nur dann , wenn es so zu tun , erzählt wird.

Es ist auch wichtig zu beachten, dass die Namen der Eingaben den Namen der Schlüssel im Personenobjekt entsprechen. Auf diese Weise können wir den Wert in Form eines Wörterbuchs wie hier dargestellt erfassen.

```
handleChange(event) {
    let person = this.state.person;
    person[event.target.name] = event.target.value;
    this.setState({person});
}
```

`person[event.target.name]` ist das gleiche wie ein `person.firstName || person.lastName` . Dies hängt natürlich davon ab, welche Eingabe gerade eingegeben wird. Da wir nicht wissen, wo der Benutzer tippen wird, können wir die Benutzereingabe-Nr. Verwenden, indem Sie ein Wörterbuch verwenden und die Namen der Eingaben mit den Namen der Tasten abgleichen egal woher der onChange aufgerufen wird.

Reaktionsformen online lesen: https://riptutorial.com/de/reactjs/topic/8047/reaktionsformen

Kapitel 25: Reaktives Routing

Examples

Beispiel für die Datei Routes.js, gefolgt von der Verwendung von Router Link in der Komponente

Platzieren Sie eine Datei wie die folgende in Ihrem obersten Verzeichnis. Es definiert, welche Komponenten für welche Pfade gerendert werden sollen

```
import React from 'react';
import { Route, IndexRoute } from 'react-router';
import New from './containers/new-post';
import Show from './containers/show';

import Index from './containers/home';
import App from './components/app';

export default(
  <Route path="/" component={App}>
    <IndexRoute component={Index} />
    <Route path="posts/new" component={New} />
    <Route path="posts/:id" component={Show} />

  </Route>
);
```

Jetzt in Ihrer obersten Ebene index.js, dem Einstiegspunkt in die App, müssen Sie diese Routerkomponente nur wie folgt rendern:

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Router, browserHistory } from 'react-router';
// import the routes component we created in routes.js
import routes from './routes';


// entry point
ReactDOM.render(
    <Router history={browserHistory} routes={routes} />
  , document.getElementById('main'));
```

Jetzt müssen Sie in Ihrer gesamten Anwendung nur `Link` anstelle von `<a>` -Tags verwenden. Bei Verwendung von Link wird mit React Router kommuniziert, um die React Router-Route in den angegebenen Link zu ändern, der wiederum die korrekte Komponente wie in routes.js definiert wiedergibt

```
import React from 'react';
import { Link } from 'react-router';
```

```
export default function PostButton(props) {
  return (
    <Link to={`posts/${props.postId}`}>
      <div className="post-button" >
        {props.title}
        <span>{props.tags}</span>
      </div>
    </Link>
  );
}
```

Routing async reagieren

```
import React from 'react';
import { Route, IndexRoute } from 'react-router';

import Index from './containers/home';
import App from './components/app';

//for single Component lazy load use this
const ContactComponent = () => {
  return {
      getComponent: (location, callback)=> {
        require.ensure([], require => {
          callback(null, require('./components/Contact')["default"]);
        }, 'Contact');
      }
    }
};

//for multiple componnets
 const groupedComponents = (pageName) => {
  return {
      getComponent: (location, callback)=> {
        require.ensure([], require => {
          switch(pageName){
            case 'about' :
                callback(null, require( "./components/about" )["default"]);
                    break ;
            case 'tos' :
                callback(null, require( "./components/tos" )["default"]);
                    break ;
          }
        }, "groupedComponents");
      }
    }
};
export default(
  <Route path="/" component={App}>
    <IndexRoute component={Index} />
    <Route path="/contact" {...ContactComponent()} />
    <Route path="/about" {...groupedComponents('about')} />
    <Route path="/tos" {...groupedComponents('tos')} />
  </Route>
);
```

Reaktives Routing online lesen: https://riptutorial.com/de/reactjs/topic/6096/reaktives-routing

Kapitel 26: Requisiten reagieren

Bemerkungen

HINWEIS: Ab React 15.5 und älter befindet sich die PropTypes-Komponente in einem eigenen npm-Paket, nämlich 'prop-types', und benötigt bei Verwendung von PropTypes eine eigene Importanweisung. Die offizielle Dokumentation zu den Änderungen finden Sie unter: https://facebook.github.io/react/blog/2017/04/07/react-v15.5.0.html

Examples

Einführung

`props` werden verwendet, um Daten und Methoden von einer übergeordneten Komponente an eine untergeordnete Komponente zu übergeben.

Interessantes über `props`

1. Sie sind unveränderlich.
2. Sie ermöglichen es uns, wiederverwendbare Komponenten zu erstellen.

Grundlegendes Beispiel

```
class Parent extends React.Component{
  doSomething(){
     console.log("Parent component");
  }
  render() {
    return <div>
         <Child
           text="This is the child number 1"
           title="Title 1"
           onClick={this.doSomething} />
         <Child
           text="This is the child number 2"
           title="Title 2"
           onClick={this.doSomething} />
      </div>
  }
}

class Child extends React.Component{
  render() {
    return <div>
       <h1>{this.props.title}</h1>
       <h2>{this.props.text}</h2>
      </div>
  }
}
```

Wie Sie im Beispiel sehen können, können wir dank `props` wiederverwendbare Komponenten erstellen.

Standard-Requisiten

`defaultProps` können Sie standardmäßig oder Ausweich - Werte für die Komponente setzen `props` . `defaultProps` ist hilfreich, wenn Sie Komponenten aus verschiedenen Ansichten mit festen Requisiten aufrufen, in einigen Ansichten müssen Sie jedoch andere Werte übergeben.

Syntax

ES5

```
var MyClass = React.createClass({
  getDefaultProps: function() {
    return {
      randomObject: {},
      ...
    };
  }
}
```

ES6

```
class MyClass extends React.Component {...}

MyClass.defaultProps = {
    randomObject: {},
    ...
}
```

ES7

```
class MyClass extends React.Component {
    static defaultProps = {
        randomObject: {},
        ...
    };
}
```

Das Ergebnis von `getDefaultProps()` oder `defaultProps` wird zwischengespeichert und verwendet, um sicherzustellen, dass `this.props.randomObject` einen Wert hat, wenn es nicht von der übergeordneten Komponente angegeben wurde.

PropTypes

`propTypes` können Sie festlegen , welche `props` Ihre Komponente Bedürfnisse und die Art sie sein sollten. Ihre Komponente funktioniert ohne Festlegen von `propTypes` . Es ist jedoch `propTypes` , diese zu definieren, da sie Ihre Komponente lesbarer macht und anderen Entwicklern, die Ihre

Komponente lesen, als Dokumentation dient. Während der Entwicklung wird React Sie warnen, wenn Sie dies versuchen Legen Sie eine Eigenschaft fest, die sich von der Definition unterscheidet, die Sie für sie definiert haben.

Einige primitive `propTypes` und häufig verwendbare `propTypes` sind -

```
  optionalArray: React.PropTypes.array,
  optionalBool: React.PropTypes.bool,
  optionalFunc: React.PropTypes.func,
  optionalNumber: React.PropTypes.number,
  optionalObject: React.PropTypes.object,
  optionalString: React.PropTypes.string,
  optionalSymbol: React.PropTypes.symbol
```

Wenn Sie " `isRequired` an einen beliebigen `propType` anhängen, muss diese `propType` beim Erstellen der Instanz dieser Komponente angegeben werden. Wenn Sie die **erforderlichen** `propTypes` nicht `propTypes` Komponenteninstanz nicht erstellt werden.

Syntax

ES5

```
var MyClass = React.createClass({
  propTypes: {
    randomObject: React.PropTypes.object,
    callback: React.PropTypes.func.isRequired,
    ...
  }
}
```

ES6

```
class MyClass extends React.Component {...}

MyClass.propTypes = {
    randomObject: React.PropTypes.object,
    callback: React.PropTypes.func.isRequired,
    ...
};
```

ES7

```
class MyClass extends React.Component {
     static propTypes = {
        randomObject: React.PropTypes.object,
        callback: React.PropTypes.func.isRequired,
        ...
    };
}
```

Komplexere Requisitenvalidierung

Auf dieselbe Weise können Sie mit `PropTypes` komplexere Validierungen angeben

Objekt überprüfen

```
...
    randomObject: React.PropTypes.shape({
        id: React.PropTypes.number.isRequired,
        text: React.PropTypes.string,
    }).isRequired,
...
```

Überprüfung auf Array von Objekten

```
...
    arrayOfObjects: React.PropTypes.arrayOf(React.PropTypes.shape({
        id: React.PropTypes.number.isRequired,
        text: React.PropTypes.string,
    })).isRequired,
...
```

Requisiten mit Spread-Operator weitergeben

Anstatt

```
var component = <Component foo={this.props.x} bar={this.props.y} />;
```

Wo jede Eigenschaft als Einzelstütze Wert übergeben werden muss , können Sie den Spread -Operator verwenden `...` für Arrays in ES6 unterstützt alle Werte weitergeben. Die Komponente wird jetzt so aussehen.

```
var component = <Component {...props} />;
```

Denken Sie daran, dass die Eigenschaften des Objekts, das Sie übergeben, auf die Requisiten der Komponente kopiert werden.

Die Reihenfolge ist wichtig. Spätere Attribute überschreiben vorherige.

```
var props = { foo: 'default' };
var component = <Component {...props} foo={'override'} />;
console.log(component.props.foo); // 'override'
```

Ein anderer Fall ist, dass Sie den Spread-Operator auch verwenden können, um nur Teile von Requisiten an untergeordnete Komponenten zu übergeben. Anschließend können Sie die Destruktursyntax von Requisiten erneut verwenden.

Es ist sehr nützlich, wenn Kinderkomponenten viele Requisiten benötigen, sie jedoch nicht einzeln übergeben werden sollen.

```
const { foo, bar, other } = this.props // { foo: 'foo', bar: 'bar', other: 'other' };
var component = <Component {...{foo, bar}} />;
const { foo, bar } = component.props
console.log(foo, bar); // 'foo bar'
```

Requisiten.Kinder und Komponentenzusammensetzung

Die "Kinderkomponenten" einer Komponente sind auf einer speziellen Requisite `props.children` . Diese Requisite ist sehr nützlich für "Compositing" -Komponenten und kann JSX-Markup intuitiver machen oder die beabsichtigte endgültige Struktur des DOM widerspiegeln:

```
var SomeComponent = function () {
    return (
        <article className="textBox">
            <header>{this.props.heading}</header>
            <div className="paragraphs">
                {this.props.children}
            </div>
        </article>
    );
}
```

Dadurch können Sie bei der späteren Verwendung der Komponente beliebig viele Unterelemente hinzufügen:

```
var ParentComponent = function () {
    return (
        <SomeComponent heading="Amazing Article Box" >
            <p className="first"> Lots of content </p>
            <p> Or not </p>
        </SomeComponent>
    );
}
```

Props.children können auch von der Komponente manipuliert werden. Da props.children ein Array sein kann oder nicht , stellt React als React.Children Dienstfunktionen bereit . Betrachten Sie im vorherigen Beispiel, ob wir jeden Absatz in ein eigenes `<section>` -Element einschließen wollten:

```
var SomeComponent = function () {
    return (
        <article className="textBox">
            <header>{this.props.heading}</header>
            <div className="paragraphs">
                {React.Children.map(this.props.children, function (child) {
                    return (
                        <section className={child.props.className}>
                            React.cloneElement(child)
                        </section>
                    );
                })}
            </div>
        </article>
    );
}
```

Beachten Sie die Verwendung von React.cloneElement, um die Requisiten aus dem untergeordneten `<p>` -Tag zu entfernen. Da Requisiten unveränderlich sind, können diese Werte nicht direkt geändert werden. Stattdessen muss ein Klon ohne diese Requisiten verwendet werden.

Wenn Sie Elemente in Schleifen hinzufügen, sollten Sie sich darüber im Klaren sein, wie React Kinder während eines Wiederherstellungsprozesses abgeglichen wird , und es sollte in Erwägung gezogen werden, eine global eindeutige `key` untergeordnete Elemente zu verwenden, die in einer Schleife hinzugefügt werden.

Ermitteln des Typs der untergeordneten Komponenten

Manchmal ist es sehr nützlich, den Typ der untergeordneten Komponente beim Durchlaufen dieser Komponenten zu kennen. Um die untergeordneten Komponenten zu durchlaufen, können Sie die React `Children.map` util-Funktion verwenden:

```
React.Children.map(this.props.children, (child) => {
  if (child.type === MyComponentType) {
    ...
  }
});
```

Das untergeordnete Objekt macht die `type` Eigenschaft verfügbar, die Sie mit einer bestimmten Komponente vergleichen können.

Requisiten reagieren online lesen: https://riptutorial.com/de/reactjs/topic/2749/requisiten-reagieren

Kapitel 27: Tasten reagieren

Einführung

Die Tasten in rea werden verwendet, um intern eine Liste von DOM-Elementen aus derselben Hierarchie zu identifizieren.

Wenn Sie also ein Array durchlaufen, um eine Liste von li-Elementen anzuzeigen, benötigt jedes li-Element einen eindeutigen Bezeichner, der in der key-Eigenschaft angegeben ist. Dies kann normalerweise die ID Ihres Datenbankelements oder der Index des Arrays sein.

Bemerkungen

Die Verwendung des Array-Index als Schlüssel wird im Allgemeinen nicht empfohlen, wenn sich das Array im Laufe der Zeit ändert. Aus den React Docs:

> Als letzten Ausweg können Sie den Index des Elements im Array als Schlüssel übergeben. Dies kann gut funktionieren, wenn die Elemente nie neu angeordnet werden, aber die Reihenfolge wird langsam.

Ein gutes Beispiel dazu: https://medium.com/@robinpokorny/index-as-a-key-is-an-anti-pattern-e0349aece318

Examples

Verwenden der ID eines Elements

Hier haben wir eine Liste mit Aufgaben, die an die Requisiten unserer Komponente übergeben werden.

Jedes ToDo-Element verfügt über eine Text- und ID-Eigenschaft. Stellen Sie sich vor, dass die id -Eigenschaft aus einem Backend-Datastore stammt und ein eindeutiger numerischer Wert ist:

```
todos = [
  {
    id: 1,
    text: 'value 1'
  },
  {
    id: 2,
    text: 'value 2'
  },
  {
    id: 3,
    text: 'value 3'
  },
  {
    id: 4,
```

```
    text: 'value 4'
  },
];
```

Wir setzen das Schlüsselattribut jedes iterierten Listenelements auf `todo-${todo.id}` damit es von `todo-${todo.id}` intern identifiziert werden kann:

```
render() {
  const { todos } = this.props;
  return (
    <ul>
      { todos.map((todo) =>
        <li key={ `todo-${todo.id}` }>
          { todo.text }
        </li>
      }
    </ul>
  );
}
```

Verwenden des Array-Indexes

Wenn Sie keine eindeutigen Datenbank-IDs zur Hand haben, können Sie auch den numerischen Index Ihres Arrays wie folgt verwenden:

```
render() {
  const { todos } = this.props;
  return (
    <ul>
      { todos.map((todo, index) =>
        <li key={ `todo-${index}` }>
          { todo.text }
        </li>
      }
    </ul>
  );
}
```

Tasten reagieren online lesen: https://riptutorial.com/de/reactjs/topic/9805/tasten-reagieren

Kapitel 28: Werkzeuge reagieren

Examples

Links

Orte, an denen React-Komponenten und -Bibliotheken zu finden sind;

- Katalog der Reaktionskomponenten
- JS.coach

Werkzeuge reagieren online lesen: https://riptutorial.com/de/reactjs/topic/6595/werkzeuge-reagieren

Kapitel 29: Wie und warum werden Schlüssel in React verwendet?

Einführung

Wenn Sie eine Liste von React-Komponenten rendern, muss jede Komponente über ein `key` verfügen. Der Schlüssel kann ein beliebiger Wert sein, muss jedoch für diese Liste eindeutig sein.

Wenn React Änderungen an einer Liste von Elementen wiedergeben muss, durchläuft React beide Listen von Kindern zur gleichen Zeit und generiert eine Mutation, wenn ein Unterschied besteht. Wenn für die Kinder keine Schlüssel festgelegt sind, durchsucht React jedes Kind. Andernfalls vergleicht React die Schlüssel, um festzustellen, welche der Liste hinzugefügt oder aus ihr entfernt wurden

Bemerkungen

Weitere Informationen finden Sie unter diesem Link, um zu erfahren, wie Sie Schlüssel verwenden: https://facebook.github.io/react/docs/lists-and-keys.html

Besuchen Sie diesen Link, um zu erfahren, warum die Verwendung von Schlüsseln empfohlen wird: https://facebook.github.io/react/docs/reconciliation.html#recursing-on-children

Examples

Basisbeispiel

Für eine klassenlose React-Komponente:

```
function SomeComponent(props){

    const ITEMS = ['cat', 'dog', 'rat']
    function getItemsList(){
        return ITEMS.map(item => <li key={item}>{item}</i>);
    }

    return (
        <ul>
            {getItemsList()}
        </ul>
    );
}
```

Für dieses Beispiel wird die obige Komponente wie folgt aufgelöst:

```
<ul>
    <li key='cat'>cat</li>
    <li key='dog'>dog</li>
```

```
    <li key='rat'>rat</li>
<ul>
```

Wie und warum werden Schlüssel in React verwendet? online lesen:
https://riptutorial.com/de/reactjs/topic/9665/wie-und-warum-werden-schlussel-in-react-verwendet-

Kapitel 30: Zustand in Reaktion

Examples

Grundzustand

Status in React-Komponenten ist für die Verwaltung und Kommunikation von Daten in Ihrer Anwendung unerlässlich. Es wird als JavaScript - Objekt dargestellt und hat *Komponentenebene* Umfang, kann es als die privaten Daten Ihrer Komponente gedacht werden.

Im folgenden Beispiel definieren wir einen Anfangszustand in der `constructor` unserer Komponente und verwenden diesen in der `render` .

```
class ExampleComponent extends React.Component {
  constructor(props){
    super(props);

    // Set-up our initial state
    this.state = {
      greeting: 'Hiya Buddy!'
    };
  }

  render() {
    // We can access the greeting property through this.state
    return(
      <div>{this.state.greeting}</div>
    );
  }
}
```

setState ()

Die `setState()` , mit der Sie Aktualisierungen der Benutzeroberfläche für Ihre React-Anwendungen vornehmen, ist der Aufruf der Funktion `setState()` . Diese Funktion führt eine *flache Verschmelzung* zwischen dem von Ihnen angegebenen neuen Status und dem vorherigen Status durch und löst ein erneutes Rendern Ihrer Komponente und aller Dekadenten aus.

Parameter

1. `updater` : Es kann sich um ein Objekt mit einer Reihe von Schlüssel-Wert-Paaren handeln, die mit dem Status zusammengeführt werden sollten, oder eine Funktion, die ein solches Objekt zurückgibt.
2. `callback (optional)` : Eine Funktion, die ausgeführt wird, nachdem `setState()` erfolgreich ausgeführt wurde. Aufgrund der Tatsache, dass Aufrufe von `setState()` von React nicht als atomar garantiert werden, kann dies manchmal nützlich sein, wenn Sie eine Aktion ausführen möchten, nachdem Sie sich `setState()` sind, dass `setState()` erfolgreich ausgeführt wurde.

Verwendungszweck:

Die `setState` Methode akzeptiert ein `updater` Argument, bei dem es sich entweder um ein Objekt mit einer Anzahl von Schlüssel-Wert-Paaren handeln kann, das in den Status `prevState` werden soll, oder um eine Funktion, die ein solches Objekt `prevState` aus `prevState` und `props` berechnet wurde.

`setState()` mit einem Objekt als `updater`

```
//
// An example ES6 style component, updating the state on a simple button click.
// Also demonstrates where the state can be set directly and where setState should be used.
//
class Greeting extends React.Component {
    constructor(props) {
        super(props);
        this.click = this.click.bind(this);
        // Set initial state (ONLY ALLOWED IN CONSTRUCTOR)
        this.state = {
            greeting: 'Hello!'
        };
    }
    click(e) {
        this.setState({
              greeting: 'Hello World!'
        });
    }
    render() {
        return(
            <div>
                <p>{this.state.greeting}</p>
                <button onClick={this.click}>Click me</button>
            </div>
        );
    }

}
```

`setState()` mit einer Funktion als `updater`

```
//
// This is most often used when you want to check or make use
// of previous state before updating any values.
//

this.setState(function(previousState, currentProps) {
  return {
    counter: previousState.counter + 1
  };
});
```

Dies kann sicherer sein als die Verwendung eines Objektarguments, bei dem mehrere Aufrufe von `setState()` verwendet werden, da mehrere Aufrufe von React zusammengestellt und gleichzeitig

ausgeführt werden können. Dies ist der bevorzugte Ansatz bei der Verwendung aktueller Requisiten, um den Status festzulegen.

```
this.setState({ counter: this.state.counter + 1 });
this.setState({ counter: this.state.counter + 1 });
this.setState({ counter: this.state.counter + 1 });
```

Diese Aufrufe können von React mithilfe von `Object.assign()` , wodurch der Zähler um 1 statt um 3 erhöht wird.

Der funktionale Ansatz kann auch verwendet werden, um die Logik zur Einstellung von Zuständen außerhalb von Komponenten zu verschieben. Dies ermöglicht die Isolierung und Wiederverwendung der Zustandslogik.

```
// Outside of component class, potentially in another file/module

function incrementCounter(previousState, currentProps) {
    return {
        counter: previousState.counter + 1
    };
}

// Within component

this.setState(incrementCounter);
```

Aufruf von `setState()` mit einem Object und einer Callback-Funktion

```
//
// 'Hi There' will be logged to the console after setState completes
//

this.setState({ name: 'John Doe' }, console.log('Hi there'));
```

Gemeinsames Antipattern

Sie sollten `props` im `state` speichern. Es gilt als Anti-Muster . Zum Beispiel:

```
export default class MyComponent extends React.Component {
    constructor() {
        super();

        this.state = {
            url: ''
        }

        this.onChange = this.onChange.bind(this);
    }
```

```
    onChange(e) {
        this.setState({
            url: this.props.url + '/days=?' + e.target.value
        });
    }

    componentWillMount() {
        this.setState({url: this.props.url});
    }

    render() {
        return (
            <div>
                <input defaultValue={2} onChange={this.onChange} />

                URL: {this.state.url}
            </div>
        )
    }
}
```

Die Prop- `url` wird im `state` gespeichert und dann geändert. Speichern Sie stattdessen die Änderungen in einem Status und erstellen Sie dann den vollständigen Pfad mit `state` und `props` :

```
export default class MyComponent extends React.Component {
    constructor() {
        super();

        this.state = {
            days: ''
        }

        this.onChange = this.onChange.bind(this);
    }

    onChange(e) {
        this.setState({
            days: e.target.value
        });
    }

    render() {
        return (
            <div>
                <input defaultValue={2} onChange={this.onChange} />

                URL: {this.props.url + '/days?=' + this.state.days}
            </div>
        )
    }
}
```

Dies liegt daran, dass wir in einer React-Anwendung eine einzige Quelle der Wahrheit haben möchten - dh für alle Daten sind nur eine Komponente und nur eine Komponente verantwortlich. Es liegt in der Verantwortung dieser Komponente, die Daten in ihrem Status zu speichern und sie über Requisiten an andere Komponenten zu verteilen.

Im ersten Beispiel behalten sowohl die MyComponent-Klasse als auch ihre übergeordnete Klasse

'url' in ihrem Status bei. Wenn wir state.url in MyComponent aktualisieren, werden diese Änderungen nicht im übergeordneten Element angezeigt. Wir haben unsere einzige Quelle der Wahrheit verloren, und es wird immer schwieriger, den Datenfluss durch unsere Anwendung zu verfolgen. Im Vergleich zum zweiten Beispiel - url wird nur im Zustand der übergeordneten Komponente beibehalten und in MyComponent als Requisite verwendet - behalten wir daher eine einzige Wahrheitsquelle bei.

Status, Ereignisse und verwaltete Steuerelemente

Hier ist ein Beispiel einer React-Komponente mit einem "verwalteten" Eingabefeld. Wenn sich der Wert des Eingabefelds ändert, wird ein Event-Handler aufgerufen, der den Status der Komponente mit dem neuen Wert des Eingabefelds aktualisiert. Der Aufruf von `setState` im Event-Handler löst einen Aufruf aus, `render` die Komponente im Dom zu aktualisieren.

```
import React from 'react';
import {render} from 'react-dom';


class ManagedControlDemo extends React.Component {

  constructor(props){
    super(props);
    this.state = {message: ""};
  }

  handleChange(e){
    this.setState({message: e.target.value});
  }

  render() {
    return (
      <div>
        <legend>Type something here</legend>
          <input
            onChange={this.handleChange.bind(this)}
            value={this.state.message}
            autoFocus />
        <h1>{this.state.message}</h1>
      </div>
   );
  }
}


render(<ManagedControlDemo/>, document.querySelector('#app'));
```

Es ist sehr wichtig, das Laufzeitverhalten zu beachten. Jedes Mal, wenn ein Benutzer den Wert im Eingabefeld ändert

- `handleChange` wird aufgerufen und so
- `setState` wird aufgerufen und so
- `render` wird aufgerufen

Pop-Quiz, nachdem Sie ein Zeichen in das Eingabefeld eingegeben haben, dessen DOM-

Elemente geändert werden

1. all dies - das div der obersten Ebene, die Legende, die Eingabe, h1
2. nur die eingabe und h1
3. nichts
4. was ist ein DOM?

Sie können hier mehr experimentieren, um die Antwort zu finden

Zustand in Reaktion online lesen: https://riptutorial.com/de/reactjs/topic/1816/zustand-in-reaktion

Kapitel 31: Zustandslose Funktionskomponenten

Bemerkungen

Zustandslose Funktionskomponenten in React sind reine Funktionen der in `props` . Diese Komponenten sind nicht abhängig vom Status und verwerfen die Verwendung von Komponentenlebenszyklusmethoden. Sie können jedoch immer noch `propTypes` und `defaultPropts` definieren.

Weitere Informationen zu statuslosen Funktionskomponenten finden Sie unter https://facebook.github.io/react/docs/reusable-components.html#stateless-functions .

Examples

Zustandslose Funktionskomponente

Mit Komponenten können Sie die Benutzeroberfläche in *unabhängige* , *wiederverwendbare* Teile aufteilen. Das ist die Schönheit von React; Wir können eine Seite in viele kleine wiederverwendbare **Komponenten** aufteilen.

Vor React v14 konnten wir eine Stateful-React-Komponente mit `React.Component` (in ES6) oder `React.createClass` (in ES5) erstellen, unabhängig davon, ob zur Verwaltung von Daten ein Status erforderlich ist oder nicht.

React v14 führte eine einfachere Methode zur Definition von Komponenten ein, die normalerweise als **zustandslose Funktionskomponenten bezeichnet werden** . Diese Komponenten verwenden einfache JavaScript-Funktionen.

Zum Beispiel:

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

Diese Funktion ist eine gültige React-Komponente, da sie ein einzelnes `props` Objektargument mit Daten akzeptiert und ein React-Element zurückgibt. Wir bezeichnen solche Komponenten als **funktional,** weil sie buchstäblich JavaScript- *Funktionen sind* .

Zustandslose Funktionskomponenten konzentrieren sich normalerweise auf die Benutzeroberfläche. state sollte von übergeordneten "Container" -Komponenten oder über Flux / Redux usw. verwaltet werden. Statuslose Funktionskomponenten unterstützen keine Status- oder Lebenszyklusmethoden.

Leistungen:

1. Kein Unterrichtsaufwand
2. Machen Sie sich keine Sorgen um `this` Keyword
3. Einfach zu schreiben und leicht zu verstehen
4. Sie müssen sich keine Sorgen um die Verwaltung der Staatswerte machen
5. Leistungsverbesserung

Zusammenfassung : Wenn Sie eine React-Komponente schreiben, für die kein Status erforderlich ist, und eine wiederverwendbare Benutzeroberfläche erstellen möchte, können Sie sie anstelle einer Standard-React-Komponente als **zustandslose Funktionskomponente** schreiben.

Nehmen wir ein einfaches Beispiel:

Nehmen wir an, wir haben eine Seite, die einen Benutzer registrieren, nach registrierten Benutzern suchen oder eine Liste aller registrierten Benutzer anzeigen kann.

Dies ist der Einstiegspunkt der Anwendung `index.js` :

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

import HomePage from './homepage'

ReactDOM.render(
    <HomePage/>,
    document.getElementById('app')
);
```

Die `HomePage` Komponente bietet die Benutzeroberfläche zum Registrieren und Suchen nach Benutzern. Beachten Sie, dass es sich um eine typische React-Komponente handelt, die Status, Benutzeroberfläche und Verhaltenscode enthält. Die Daten für die Liste der registrierten Benutzer werden in dem gespeicherten `state` variabel, aber unsere wiederverwendbare `List` (siehe unten) kapselt den UI - Code für die Liste.

`homepage.js` :

```
import React from 'react'
import {Component} from 'react';

import List from './list';

export default class Temp extends Component{

    constructor(props) {
        super();
        this.state={users:[], showSearchResult: false, searchResult: []};
    }

    registerClick(){
        let users = this.state.users.slice();
        if(users.indexOf(this.refs.mail_id.value) == -1){
            users.push(this.refs.mail_id.value);
            this.refs.mail_id.value = '';
            this.setState({users});
        }else{
```

```
            alert('user already registered');
        }
    }

    searchClick(){
        let users = this.state.users;
        let index = users.indexOf(this.refs.search.value);
        if(index >= 0){
            this.setState({searchResult: users[index], showSearchResult: true});
        }else{
            alert('no user found with this mail id');
        }
    }

    hideSearchResult(){
        this.setState({showSearchResult: false});
    }

    render() {
        return (
            <div>
                <input placeholder='email-id' ref='mail_id'/>
                <input type='submit' value='Click here to register'
onClick={this.registerClick.bind(this)}/>
                <input style={{marginLeft: '100px'}} placeholder='search' ref='search'/>
                <input type='submit' value='Click here to register'
onClick={this.searchClick.bind(this)}/>
                {this.state.showSearchResult ?
                    <div>
                        Search Result:
                        <List users={[this.state.searchResult]}/>
                        <p onClick={this.hideSearchResult.bind(this)}>Close this</p>
                    </div>
                    :
                    <div>
                        Registered users:
                        <br/>
                        {this.state.users.length ?
                            <List users={this.state.users}/>
                            :
                            "no user is registered"
                        }
                    </div>
                }
            </div>
        );
    }
}
```

Schließlich zeigt unsere **stateless-Funktionskomponente** `List` , die verwendet wird, sowohl die Liste der registrierten Benutzer *als auch* die Suchergebnisse, ohne jedoch den Status selbst beizubehalten.

`list.js` :

```
import React from 'react';
var colors = ['#6A1B9A', '#76FF03', '#4527A0'];

var List = (props) => {
```

```
    return(
        <div>
            {
                props.users.map((user, i)=>{
                    return(
                        <div key={i} style={{color: colors[i%3]}}>
                            {user}
                        </div>
                    );
                })
            }
        </div>
    );
}

export default List;
```

Referenz: https://facebook.github.io/react/docs/components-and-props.html

Zustandslose Funktionskomponenten online lesen:
https://riptutorial.com/de/reactjs/topic/6588/zustandslose-funktionskomponenten

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit React	Adam, Adrián Daraš, Alex, Alex Young, Anuj, Bart Riordan, Cassidy, Community, Daksh Gupta, Dave Kaye, diabolicfreak, DMan, Donald, Everettss, Gianluca Esposito, himanshuIIITian , hyde, Ilya Lyamkin, Inanc Gumus, ivarni, jengeb, jolyonruss, Jon Chan, JordanHendrix, juandemarco, Kaloyan Kosev, Konstantin Grushetsky, Maksim, Marty, MaxPRafferty, Md. Nahiduzzaman Rose, Md.Sifatul Islam, Ming Soon, MMachinegun, Nick Bartlett, orvi, paqash, Prakash, rossipedia , Shabin Hashim, Simplans, Sunny R Gupta, TheShadowbyte, Timo, Tushar Khanna, user2314737
2	AJAX-Anruf reagieren	adamboro, Fabian Schultz, Jason Bourne, lifeiscontent, McGrady, Sunny R Gupta
3	Boilerplate reagieren [Reagieren + Babel + Webpack]	Mihir, parlad neupane, Tien Do
4	Einführung in das serverseitige Rendering	Adrián Daraš, MauroPorrasP
5	Einrichten der React-Umgebung	ghostffcode, Tien Do
6	Einrichten eines grundlegenden Webpacks, Reagieren und Babel-Umgebung	Bart Riordan, Tien Do, Zac Braddy
7	Formulare und Benutzereingaben	Everettss, Henrik Karlsson, ivarni, Timo
8	Installation	Rene R, Ruairi O'Brien
9	Installation von React, Webpack & Typescript	Aron
10	JSX	Kaloyan Kosev, Ming Soon
11	Kommunikation zwischen	David, Kaloyan Kosev

	Komponenten	
12	Kommunizieren Sie zwischen Komponenten	Random User
13	Komponenten	akashrajkn, Anuj, Bart Riordan, Bond, Brandon Roberts, Denis Ivanov, Diego V, DMan, Evan Hammer, Everettss, goldbullet, GordyD, hmnzr, Ilya Lyamkin, ivarni, Jagadish Upadhyay, jbmartinez, John Ruddell, jolyonruss, Jon Chan, jonathangoodman, JordanHendrix, justabuzz, k170, Kousha, Kyle Richardson, m_callens, Maayan Glikser, Michael Peyper, Paul Graffam, philpee2, QoP, Radu Brehar, Sai Vikas, sjmarshy, Timo, Vlad Bezden, WooCaSh, Zakaria Ridouh, zurfyx
14	Komponenten höherer Ordnung	Dennis Stücken
15	Lebenszyklus der Reaktionskomponente	Alex Young, Alexg2195, Anuj, Ashari, Everettss, F. Kauder, irrigator, John Ruddell, QoP, Salman Saleem, Saravana, Siddharth, skav, Timo, ultrasamad, Vivian, WitVault
16	Lösungen für die Benutzeroberfläche	vintproykt
17	Performance	Aditya Singh, lustoykov, thibmaek
18	React with Flow verwenden	JimmyLv, lifeiscontent, Rifat, Rory O'Kane
19	React.createClass vs erweitert React.Component	Kaloyan Kosev, leonardoborges, Michael Peyper, pwolaq, Qianyue, sqzaman
20	ReactJS auf Flux-Weise verwenden	vintproykt
21	ReactJS mit jQuery verwenden	Kousha, Shuvo Habib
22	ReactJS mit Typescript verwenden	Everettss, John Ruddell, kevgathuku, Leone, Rajab Shakirov
23	Reagiere mit Redux	Jim
24	Reaktionsformen	promisified
25	Reaktives Routing	abhirathore2006, Robeen

26	Requisiten reagieren	Ahmad, Anuj, Danillo Corvalan, Everettss, Faktor 10, Fellow Stranger, hansn, Ilya Lyamkin, Jack7, Jagadish Upadhyay, JimmyLv, MaxPRafferty, QoP, Sergii Bishyr, vintproykt, WitVault, zbynour
27	Tasten reagieren	Dennis Stücken, thibmaek
28	Werkzeuge reagieren	brillout
29	Wie und warum werden Schlüssel in React verwendet?	Sammy I.
30	Zustand in Reaktion	Alex Young, Alexander, Brad Colthurst, Everettss, Kousha, Kyle Richardson, QoP, skav, Timo
31	Zustandslose Funktionskomponenten	Adam, Mark Lapierre, Mayank Shukla, Valter Júnior