by robin wieruch

the Road to learn React

The Road to learn React

Your journey to master plain yet pragmatic React

Robin Wieruch

This book is for sale at http://leanpub.com/the-road-to-learn-react

This version was published on 2019-03-09

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2016 - 2019 Robin Wieruch

Tweet This Book!

Please help Robin Wieruch by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I am going to learn #ReactJs with The Road to learn React by @rwieruch Join me on my journey https://roadtoreact.com

The suggested hashtag for this book is #ReactJs.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#ReactJs

Contents

Foreword . . . . . **i**
About the Author ii
Testimonials iii
Education for Children iv
FAQ v
Change Log vii
Challenge ix

Introduction to React **1**
Hi, my name is React. 2
Requirements 4
Installation 7
Zero-Configuration Setup 8
Introduction to JSX 11
ES6 const and let 14
ReactDOM 16
Hot Module Replacement 17
Complex JavaScript in JSX 19
ES6 Arrow Functions 23
ES6 Classes 25

Basics in React **28**
Local Component State 29
ES6 Object Initializer 32
Unidirectional Data Flow 34
Bindings 39
Event Handler 44
Interactions with Forms and Events 49
ES6 Destructuring 56
Controlled Components 59
Split Up Components 61
Composable Components 64
Reusable Components 66
Component Declarations 69

Styling Components 72

Getting Real with APIs **79**
Lifecycle Methods 80
Fetching Data 83
ES6 Spread Operators 87
Conditional Rendering 90
Client- or Server-side Search 93
Paginated Fetch 98
Client Cache 102
Error Handling 109
Axios instead of Fetch 113

Code Organization and Testing **117**
ES6 Modules: Import and Export 118
Code Organization with ES6 Modules 121
Snapshot Tests with Jest 126
Unit Tests with Enzyme 133
Component Interface with PropTypes 136
Debugging with React Developer Tools 140

Advanced React Components **142**
Ref a DOM Element 143
Loading 147
Higher-Order Components 151
Advanced Sorting 155

State Management in React **168**
Lifting State 169
Revisited: setState() 176
Taming the State 181

Final Steps to Production **183**
Eject 184
Deploy your App 185

Outline **186**

Foreword

The Road to learn React teaches the fundamentals of React. You will build a real-world application in plain React without complicated tooling. Everything from project setup to deployment on a server will be explained for you. The book comes with additional referenced reading material and exercises with each chapter. After reading the book, you will be able to build your own applications in React. The material is kept up to date by myself and the community.

In the Road to learn React, I offer a foundation before you dive into the broader React ecosystem. The concepts will have less tooling and less external state management, but a lot of information about React. It explains general concepts, patterns, and best practices in a real world React application.

Essentially, you will learn to build your own React application from scratch, with features like pagination, client-side caching, and interactions like searching and sorting. Additionally, you will transition from JavaScript ES5 to JavaScript ES6. I hope this book captures my enthusiasm for React and JavaScript, and that it helps you get started with it.

About the Author

I am a German software and web engineer dedicated to learning and teaching programming in JavaScript. After obtaining my Master's Degree in computer science, I continued learning on my own. I gained experience from the startup world, where I used JavaScript intensively during both my professional life and spare time, which eventually led to a desire to teach others about these topics.

For a few years, I worked closely with an exceptional team of engineers at a company called Small Improvements, developing large scale applications. The company offered a SaaS product that enables customers to give feedback to businesses. This application was developed using JavaScript on its frontend, and Java as its backend. The first iteration of Small Improvements' frontend was written in Java with the Wicket Framework and jQuery. When the first generation of SPAs became popular, the company migrated to Angular 1.x for its frontend application. After using Angular for over two years, it became clear that Angular wasn't the best solution to work with state intense applications, so they made the jump to React and Redux. This enabled it to operate on a large scale successfully.

During my time in the company, I regularly wrote articles about web development on my website. I received great feedback from people learning from my articles which allowed me to improve my writing and teaching style. Article after article, I grew my ability to teach others. I felt that my first articles were packed with too much information, quite overwhelming for students, but I improved by focusing on one subject at a time.

Currently, I am a self-employed software engineer and educator. I find it a fulfilling pastime to see students thrive by giving them clear objectives and short feedback loops. You can find more information about me and ways to support and work with me on my website[1].

[1] https://www.robinwieruch.de/about

Testimonials

There are many testimonials[2], ratings[3] and reviews[4] about the book that you can read to ascertain its quality. I am proud of it, and I never expected such overwhelming feedback. I would love to find your rating/review. It helps me to spread the word about the book and make improvements for future projects. The following shows a short excerpt of these voices:

Muhammad Kashif[5]: "The Road to Learn React is a unique book that I recommend to any student or professional interested in learning react basics to advanced level. It is packed with insightful tips and techniques that are hard to find elsewhere, and remarkably thorough in its use of examples and references to sample problems, i have 17 years of experience in web and desktop app development, and before reading this book i was having trouble in learning react, but this book works like magic."

Andre Vargas[6]: "The Road to Learn React by Robin Wieruch is such an awesome book! Most of what I learned about React and even ES6 was through it!"

Nicholas Hunt-Walker, Instructor of Python at a Seattle Coding School[7]: "This is one of the most well-written & informative coding books I've ever worked through. A solid React & ES6 introduction."

Austin Green[8]: "Thanks, really loved the book. Perfect blend to learn React, ES6, and higher level programming concepts."

Nicole Ferguson[9]: "I'm doing Robin's Road to Learn React course this weekend & I almost feel guilty for having so much fun."

Karan[10]: "Just finished your Road to React. Best book for a beginner in the world of React and JS. Elegant exposure to ES. Kudos! :)"

Eric Priou[11]: "The Road to learn React by Robin Wieruch is a must read. Clean and concise for React and JavaScript."

[2] https://roadtoreact.com/
[3] https://www.goodreads.com/book/show/37503118-the-road-to-learn-react
[4] https://www.amazon.com/dp/B077HJFCQX
[5] https://twitter.com/appsdevpk/status/848625244956901376
[6] https://twitter.com/andrevar66/status/853789166987038720
[7] https://twitter.com/nhuntwalker/status/845730837823840256
[8] https://twitter.com/AustinGreen/status/845321540627521536
[9] https://twitter.com/nicoleffe/status/833488391148822528
[10] https://twitter.com/kvss1992/status/889197346344493056
[11] https://twitter.com/erixtekila/status/840875459730657283

Education for Children

The book should enable everyone to learn React. However, not everyone has access to the required resources, because not everyone is educated in the English language. I want to use this project to support other projects that teach children English in the developing world.

- April to 18. April, 2017, Giving Back, By Learning React[12]

[12] https://www.robinwieruch.de/giving-back-by-learning-react/

FAQ

How to get updates?

I have two channels where I share updates about my content. You can subscribe to updates by email[13] or follow me on Twitter[14]. Regardless of the channel, my objective is to only share quality content. Once you receive notification the book has changed, you can download a new version of it.

Does it use the recent React version?

The book always receives an update when the React version is updated. Programming books are usually outdated soon after their release, but since this book is self-published, I can update it as needed.

Does it cover Redux?

It doesn't, so I have written a second book. The Road to learn React should give you a solid foundation before you dive into advanced topics. The implementation of the sample application in the book will show that you don't need Redux to build an application in React. After you have read the book, you should be able to implement a solid application without Redux. Then you can read my second book, Taming the State in React, to learn Redux.

Does it use JavaScript ES6?

Yes. Don't worry, though, you will be fine if you are familiar with JavaScript ES5. All JavaScript ES6 features, that I describe in The Journey to Learn React, will transition from ES5 to ES6. The book does not only teach React, but also all useful JavaScript ES6 features.

How to get access to the source code projects and screencasts series?

If you bought one of the extended packages that grant access to the source code projects, screencast series or any other add-on, you should find these on your course dashboard[15]. If you bought the course other than the official Road to React[16] course platform, create an account on the platform, and then go to the Admin page and contact me with one of the email templates. After that I can enroll you in the course. If you haven't bought one of the extended packages, you can reach out any time to upgrade your content to access the source code projects and screencast series.

Can I get a copy of the book if I bought it on Amazon?

If you have bought the book on Amazon, you may have seen that the book is available on my website too. Since I use Amazon as one way to monetize my often free content, I honestly thank you for your support and invite you to sign up on Road to React[17]. There you can write me an email (Admin page) about your purchase, so that I can unlock the whole course package for you. In addition, you can always download the latest ebook version of the book on the platform.

[13] https://www.getrevue.co/profile/rwieruch
[14] https://twitter.com/rwieruch
[15] https://roadtoreact.com/my-courses
[16] https://roadtoreact.com
[17] https://roadtoreact.com

How can I get help while reading the book?

The book has a Slack Group[18] for people who are reading along. You can join the channel to get help, or to help others, as helping others may help you internalize your own understanding. If there is no one available to help you, you can always reach out to me.

Is there any troubleshoot area?

If you run into problems, please join the Slack Group. Also, check the open issues on GitHub[19] or in the GitHub repositories of the applications you will build along the way to see if any solutions are listed for specific issue. If your problem wasn't mentioned, open a new issue where you can explain your problem, provide a screenshot, and offer more details (e.g. book page, node version).

Can I help to improve the content?

Yes, I love to hear feedback. Simply open an issue on GitHub[20]. These can be technical improvements, or clarification on the discussed topics. You can open pull requests on the GitHub page as well.

Is there a money back guarantee?

Yes, there is 100% money back guarantee for two months if you don't think it's a good fit. Please reach out to me to get a refund.

How to support the project?

If you find my lessons useful and would like to contribute, seek my website's About Page[21] for information about how to offer support. It is also very helpful for my readers spread the word about how my books helped them, so others might discover ways to improve their web development skillsets. Contributing through any of the provided channels gives me the freedom to create in-depth courses, and to continue offering free material.

What's your motivation behind the book?

I want to teach about this topic consistently. I often find materials online that don't receive update, or only applies to a small part of a topic. Sometimes people struggle to find consistent and up-to-date resources to learn from. I want to provide this consistent and up-to-date learning experience. Also, I hope I can support the less fortunate with my projects by giving them the content for free or by having other impacts[22]. Recently I've found myself fulfilled when teaching others about programming, as it's a meaningful activity I prefer over any 9 to 5 job at any company. I hope to continue this path in the future.

[18] https://slack-the-road-to-learn-react.wieruch.com/
[19] https://github.com/rwieruch/the-road-to-learn-react/issues
[20] https://github.com/rwieruch/the-road-to-learn-react
[21] https://www.robinwieruch.de/about/
[22] https://www.robinwieruch.de/giving-back-by-learning-react/

Change Log

10. January 2017:

- v2 Pull Request[23]
- even more beginner friendly
- 37% more content
- 30% improved content
- 13 improved and new chapters
- 140 pages of learning material
- + interactive course of the book on educative.io[24]

08. March 2017:

- v3 Pull Request[25]
- 20% more content
- 25% improved content
- 9 new chapters
- 170 pages of learning material

15. April 2017:

- upgrade to React 15.5

5. July 2017:

- upgrade to node 8.1.3
- upgrade to npm 5.0.4
- upgrade to create-react-app 1.3.3

17. October 2017:

- upgrade to node 8.3.0
- upgrade to npm 5.5.1
- upgrade to create-react-app 1.4.1
- upgrade to React 16
- v4 Pull Request[26]

[23] https://github.com/rwieruch/the-road-to-learn-react/pull/18
[24] https://www.educative.io/collection/5740745361195008/5676830073815040
[25] https://github.com/rwieruch/the-road-to-learn-react/pull/34
[26] https://github.com/rwieruch/the-road-to-learn-react/pull/72

- 15% more content
- 15% improved content
- 3 new chapters (Bindings, Event Handlers, Error Handling)
- 190+ pages of learning material
- +9 Source Code Projects[27]

17. February 2018:

- upgrade to node 8.9.4
- upgrade to npm 5.6.0
- upgrade to create-react-app 1.5.1
- v5 Pull Request[28]
- more learning paths
- extra reading material
- 1 new chapter (Axios instead of Fetch)

31. August 2018:

- professional proofreading and editing by Emmanuel Stalling
- 16 Source Code Projects[29]
- v6 Pull Request[30]

3. October 2018:

- upgrade to node 10.11.0
- upgrade to npm 6.4.1
- upgrade to create-react-app 2.0.2

[27] https://roadtoreact.com
[28] https://github.com/the-road-to-learn-react/the-road-to-learn-react/pull/105
[29] https://roadtoreact.com
[30] https://github.com/the-road-to-learn-react/the-road-to-learn-react/pull/172

Challenge

Personally I write a lot about my learnings. That's how I got where I am right now. You are teaching a topic at its best when you have just learned about it yourself. Since teaching helped me a lot in my career, I want you to experience the same effects of it. But first you have to do the cause: teaching yourself. My challenge for the book is the following: teach others what you are learning while reading the book. A couple of breakpoints on how you could achieve it:

- Write a blog post about a specific topic from the book. It's not about copying and pasting the material but rather about teaching the topic your own way. Find your own words to explain something, grab a problem and solve it, and dive even more into the topic by understanding every detail about it. Then teach it to others in this one article. You will see how it fills your knowledge gaps, because you have to dig deeper into the topic, and how it opens doors for your career in the long term.
- If you are active on social media, grab a couple of things you have learned while reading the book and share them with your friends. For instance, you can tweet a hot tip on Twitter about your last learning from the book which may be interesting for others too. Just take a screenshot of the passage of the book or even better: write about it in your own words. That's how you can get into teaching without investing much time.
- If you feel confident recording your learning adventure, share your way through the book on Facebook Live, YouTube Live or Twitch. It helps you to stay concentrated and work your way through the book. Even though you don't have many people following your live session, you can always use the video to put it on YouTube afterward. Besides it is a great way to verbalize your problems and how you are going to solve them.

I would love to see people doing especially the last breakpoint: record yourself while reading this book, implementing the application(s), and conducting the exercises, and put the final version on YouTube. If parts in between of the recording are taking longer, just cut the video or use a timelapse effect for them. If you get stuck and need to fix a bug, don't leave it out but rather include these passages in the video, because they are so valuable for your audience which may run into the same issues. I believe it is important to have these parts in your video. A couple of tips for the video:

- Do it in your native language or in English if you feel comfortable with it.
- Verbalize your thoughts, the things you are doing or the problems you are running into. Having a visual video is only one part of the challenge, but the other part is you narrating through the implementation. It doesn't have to be perfect. Instead it should feel natural and not polished as all the other video courses online where nobody runs into problems.
- If you run into bugs, embrace the trouble. Try to fix the problem yourself and search online for help, don't give up, and speak about the problem and how you attempt to solve it. This helps others to follow your thought process. As I said before, it has no value to follow polished video courses online where the instructor never runs into problems. It's the most valuable part to see someone else fixing a bug in the source code.

- Some words about the technical side of the recording: Check your audio before you record a longer video. It should have the correct volume and the quality should be alright too. Regarding your editor/IDE/terminal, make sure to increase the font size. Maybe it is possible to place the code and the browser side by side. If not, make them fullscreen and switch between (e.g. MacOS CMD + Tab).
- Edit the video yourself before you put it on YouTube. It doesn't have to be a high quality, but you should try to keep it concise for your audience (e.g. leaving out the reading passages and rather summarize the steps in your own words).

In the end, you can reach out to me for promoting anything you have released. For instance, if the video turns out well, I would love to include it in this book as officially supplementary material. Just reach out to me once you finished it. After all, I hope you accept this challenge to enhance your learning experience while reading the book which also may help others. I wish you all the best for it.

Introduction to React

This chapter is an introduction to React, a JavaScript library for rendering interfaces in single-page and mobile applications, where I explain why developers should consider adding the React library to their toolbelts. We will dive into the React ecosystem, creating your first React application from scratch with no configuration. Along the way, we will introduce **JSX**, the syntax for React, and **ReactDOM**, so you have an understanding of React's practical uses in modern web applications.

Hi, my name is React.

Single page applications (SPA[31]) have become increasingly popular in recent years, as frameworks like Angular, Ember, and Backbone allow JavaScript developers to build modern web applications using techniques beyond vanilla JavaScript and jQuery. The three mentioned are among the first SPAs, each coming into its own between 2010 and 2011, but there are many more options for single-page development. The first generation of SPA frameworks arrived at the enterprise level, so their frameworks are more rigid. React, on the other hand, remains an innovative library that has been adopted by many technological leaders like Airbnb, Netflix, and Facebook[32].

React was released by Facebook's web development team in 2013 as a view library, which makes it the 'V' in the MVC[33] (model view controller). As a view, it allows you to render components as viewable elements in a browser, while its ecosystem lets us build single page applications. While the first generation of frameworks tried to solve many things at once, React is only used to build your view layer; specifically, it is a library wherein the view is a hierarchy of composable components. If you haven't heard about MVC before, don't bother about it, because it's just there to put React historically into context for people who come from other programming languages.

In React, the focus remains on the view layer until more aspects are introduced to the application. These are the building blocks for an SPA, which are essential to build a mature application. They come with two advantages:

- You can learn the building blocks one at a time without having to understand them altogether. In contrast, an SPA framework gives you every building block from the start. This book focuses on React as the first building block. More building blocks will eventually follow.
- All building blocks are interchangeable, which makes the ecosystem around React highly innovative. Multiple solutions can compete with each other, and you can choose the most appealing solution for any given challenge.

React is one of the best choices for building modern web applications. Again, it only delivers the view layer, but the surrounding ecosystem makes up an entirely flexible and interchangeable framework. React has a slim API, a robust and evolving ecosystem, and a great community.

Exercises

If you'd like to know more about why I chose React, or to find a more in-depth look at the topics mentioned above, these articles grant a deeper perspective:

- Why I moved from Angular to React[34]

[31] https://en.wikipedia.org/wiki/Single-page_application
[32] https://github.com/facebook/react/wiki/Sites-Using-React
[33] https://en.wikipedia.org/wiki/Modelâ€“viewâ€“controller
[34] https://www.robinwieruch.de/reasons-why-i-moved-from-angular-to-react/

- React's flexible ecosystem[35]
- How to learn a framework[36]

[35] https://www.robinwieruch.de/essential-react-libraries-framework/
[36] https://www.robinwieruch.de/how-to-learn-framework/

Requirements

To follow this book, you should be familiar with the basics of web development, i.e how to use HTML, CSS, and JavaScript. It also makes sense to understand how APIs[37] work, as they will be covered thoroughly. Also, I encourage you to join the official Slack Group[38] to be a part of a growing React community where you can learn from and help others.

Editor and Terminal

For the lessons, you will need a text editor or an IDE and terminal (command line tool). I have provided a setup guide[39] if you need additional help. Optionally, we recommend you keep your projects in GitHub while conducting the exercises in this book. There is a short guide[40] on how to use these tools. Github has excellent version control, so you can see what changes were made if you make a mistake or just want a more direct way to follow along.

Node and NPM

Finally, you will need an installation of node and npm[41]. Both are used to manage libraries you will need along the way. In this book, you will install external node packages via npm (node package manager). These node packages can be libraries or whole frameworks.

You can verify your versions of node and npm on the command line. If you don't get any output in the terminal, you need to install node and npm first. These are my versions at the time of writing this book:

Command Line

```
node --version
*v10.11.0
npm --version
*v6.9.0
```

The additional content of this section is a crash course in node and npm. It is not exhaustive, but it will cover all of the necessary tools. If you are familiar with both of them, you can skip this section.

The **node package manager** (npm) installs external node packages from the command line. These packages can be a set of utility functions, libraries, or whole frameworks, and they are the dependencies of your application. You can either install these packages to your global node package folder, or to your local project folder.

Global node packages are accessible from everywhere in the terminal, and only need to be installed to the global directory once. Install a global package by typing the following into a terminal:

[37] https://www.robinwieruch.de/what-is-an-api-javascript/
[38] https://slack-the-road-to-learn-react.wieruch.com/
[39] https://www.robinwieruch.de/developer-setup/
[40] https://www.robinwieruch.de/git-essential-commands/
[41] https://nodejs.org/en/

Command Line

```
npm install -g <package>
```

The `-g` flag tells npm to install the package globally. Local packages are used in your application by default. For our purposes, we will install React to the local directory terminal by typing:

Command Line

```
npm install react
```

The installed package will automatically appear in a folder called *node_modules/* and will be listed in the *package.json* file next to your other dependencies.

To initialize the *node_modules/* folder and the *package.json* file for your project, use the following npm command. Then, you can install new local packages via npm:

Command Line

```
npm init -y
```

The `-y` flag initializes all the defaults in your *package.json*. After initializing your npm project, you are ready to install new packages via `npm install <package>`.

The *package.json* file allows you to share your project with other developers without sharing all the node packages. It will contain references to all node packages used in your project, called **dependencies**. Other users can copy a project without the dependencies using the references in *package.json*, where the references make it easy to install all packages using `npm install`. A `npm install` script will take all the dependencies listed in the *package.json* file and install them in the *node_modules/* folder.

Finally, there's one more command to cover about npm:

Command Line

```
npm install --save-dev <package>
```

The `--save-dev` flag indicates that the node package is only used in the development environment, meaning it won't be used in when the application is deployed to a the server or used in production. It is useful for testing an application using a node package, but want to exclude it from your production environment.

Some of you may want to use other package managers to work with node packages in your applications. **Yarn** is a dependency manager that works similar to **npm**. It has its own list of commands, but you still have access to the same npm registry. Yarn was created to solve issues npm couldn't, but both tools have evolved to the point where either will suffice today.

Exercises:

- Set up a throw away npm project using the terminal:
  - Create a new folder with `mkdir <folder_name>`
  - Navigate into the folder with `cd <folder_name>`
  - Execute `npm init -y` or `npm init`
  - Install a local package like React with `npm install react`
  - Check the *package.json* file and the *node_modules/* folder
  - Attempt to uninstall and reinstall the *react* node package
- Read about npm[42]
- Read about yarn[43] package manager

[42] https://docs.npmjs.com/
[43] https://yarnpkg.com/en/docs/

Installation

There are many approaches to getting started with a React application. The first we'll explore is a CDN, short for content delivery network[44]. Don't worry too much about CDNs now, because you will not use them in this book, but it makes sense to explain them briefly. Many companies use CDNs to host files publicly for their consumers. Some of these files are libraries like React, since the bundled React library is just a *react.js* JavaScript file.

To get started with React by using a CDN, find the `<script>` tag in your web page HTML that points to a CDN url. You will need two libraries: *react* and *react-dom*.

Code Playground

```
<script
  crossorigin
  src="https://unpkg.com/react@16/umd/react.development.js"
></script>

<script
  crossorigin
  src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"
></script>
```

You can also get React into your application by initializing it as node project. With a *package.json* file, you can install *react* and *react-dom* from the command line. However, the folder must be initialized as a npm project using `npm init -y` with a *package.json* file. You can install multiple node packages with npm:

Command Line

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

This approach is often used to add React to an existing application managed with npm.

You may also have to deal with Babel[45] to make your application aware of JSX (the React syntax) and JavaScript ES6. Babel transpiles your code–that is, it converts it to vanilla JavaScript–so most modern browsers can interpret JavaScript ES6 and JSX. Because of this difficult setup, Facebook introduced *create-react-app* as a zero-configuration React solution. The next section will show you how to setup your application using this bootstrapping tool.

Exercises:

- Read about React installations[46]

[44] https://en.wikipedia.org/wiki/Content_delivery_network
[45] http://babeljs.io/
[46] https://reactjs.org/docs/getting-started.html

Zero-Configuration Setup

In the Road to learn React, we will be using create-react-app[47] to bootstrap your application. It's an opinionated yet zero-configuration starter kit for React introduced by Facebook in 2016, recommended for beginners by 96% of React users[48]. In *create-react-app* the tooling and configuration evolve in the background, while the focus is on the application implementation.

To get started, install the package to your global node packages, which keeps it available on the command line to bootstrap new React applications:

Command Line

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

You can check the version of *create-react-app* to verify a successful installation on your command line:

Command Line

```
create-react-app --version
*v2.0.2
```

Now you are ready to bootstrap your first React application. The example will be referred to as *hackernews*, but you may choose any name you like. First, navigate into the folder:

Command Line

```
create-react-app hackernews
cd hackernews
```

Now you can open the application in your editor. The following folder structure, or a variation of it depending on the *create-react-app* version, should be presented to you:

Folder Structure

```
hackernews/
  README.md
  node_modules/
  package.json
  .gitignore
  public/
    favicon.ico
    index.html
    manifest.json
```

[47] https://github.com/facebookincubator/create-react-app
[48] https://twitter.com/dan_abramov/status/806985854099062785

```
src/
  App.css
  App.js
  App.test.js
  index.css
  index.js
  logo.svg
  serviceWorker.js
```

This is a breakdown of the folders and files:

- **README.md**: The .md extension indicates the file is a markdown file. Markdown is used as a lightweight markup language with plain text formatting syntax. Many source code projects come with a *README.md* file to give you initial instructions about the project. When pushing your project to a platform such as GitHub, the *README.md* file usually displays information about the content contained in the repository. Because you used *create-react-app*, your *README.md* should be the same as the official create-react-app GitHub repository[49].
- **node_modules/**: This folder contains all node packages that have been installed via npm. Since you used *create-react-app*, there should already be a couple of node modules installed for you. You will rarely touch this folder, because node packages are generally installed and uninstalled with npm from the command line.
- **package.json**: This file shows you a list of node package dependencies and other project configurations.
- **.gitignore**: This file displays all files and folders that shouldn't be added to your git repository when using git; such files and folders should only be located in your local project. The *node_modules/* folder is one example. It is enough to share the *package.json* file with others, so they can install dependencies on their end with `npm install` without your dependency folder.
- **public/**: This folder holds development files, such as *public/index.html*. The index is displayed on localhost:3000 when developing your app. The boilerplate takes care of relating this index with all the scripts in *src/*.
- **build/** This folder is created when you build the project for production, as it holds all of the production files. When building your project for production, all the source code in the *src/* and *public/* folders are bundled and placed in the build folder.
- **manifest.json** and **serviceWorker.js**: These files won't be used for this project, so you can ignore them for now.

In the beginning, everything you need is located in the *src/* folder. The main focus lies on the *src/App.js* file which is used to implement React components. It will be used to implement your application, but later you might want to split up your components into multiple files, where each file maintains one or more components on its own.

[49] https://github.com/facebookincubator/create-react-app

Additionally, you will find a *src/App.test.js* file for your tests, and a *src/index.js* as an entry point to the React world. You will get to know both files intimately in a later chapter. There is also a *src/index.css* and a *src/App.css* file to style your general application and components, which comes with the default style when you open them. You will modify them later as well.

The *create-react-app* application is a npm project you can use to install and uninstall node packages. It comes with the following npm scripts for your command line:

Command Line

```
# Runs the application in http://localhost:3000
npm start

# Runs the tests
npm test

# Builds the application for production
npm run build
```

The scripts are defined in your *package.json*, and your basic React application is bootstrapped. The following exercises will finally allow you to run your bootstrapped application in a browser.

Exercises:

- Confirm your source code for the last section[50]
  - Confirm the changes from the last section[51]
- `npm start` your application and visit the application in your browser (Exit the command by pressing Control + C)
- Run the `npm test` script
- Run the `npm run build` script and verify that a *build/* folder was added to your project (you can remove it afterward. Note that the build folder can be used later on to deploy your application[52])
- Familiarize yourself with the folder structure
- Check the content of the files
- Read about npm scripts and create-react-app[53]

[50] http://bit.ly/2Tt3Vd8
[51] http://bit.ly/2Cc5CRw
[52] https://www.robinwieruch.de/deploy-applications-digital-ocean/
[53] https://github.com/facebookincubator/create-react-app

Introduction to JSX

Now we will get to know JSX, the syntax in React. As mentioned before, *create-react-app* has already bootstrapped a basic application for you, and all files come with their own default implementations. For now, the only file we will modify is the *src/App.js* file.

src/App.js

```
import React, { Component } from 'react';
import logo from './logo.svg';
import './App.css';

class App extends Component {
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <p>
            Edit <code>src/App.js</code> and save to reload.
          </p>
          <a
            className="App-link"
            href="https://reactjs.org"
            target="_blank"
            rel="noopener noreferrer"
          >
            Learn React
          </a>
        </header>
      </div>
    );
  }
}

export default App;
```

Don't worry if you're confused by the import/export statements and class declaration now. These are features of JavaScript ES6 we will revisit in a later chapter.

In the file you should see a **React ES6 class component** with the name App. This is a component declaration. After you have declared a component, you can use it as an element anywhere in your application. It will produce an **instance** of your **component** or, in other words, the component gets instantiated.

Code Playground

```
// component declaration
class App extends Component {
  ...
}

// component usage (also called instantiation for a class)
// creates an instance of the component
<App />
```

The returned **element** is specified in the `render()` method. The components you instantiated earlier are made up of elements, so it is important to understand the differences between a component, an instance of a component, and an element.

You should see where the App component is instantiated, else you couldn't see the rendered output in a browser. The App component is only the declaration, but not the usage. You can instantiate the component anywhere in your JSX with `<App />`. You will see later where this happens in this application.

The content in the render block may look similar to HTML, but it is actually JSX. JSX allows you to mix HTML and JavaScript. It is powerful, but it can be confusing when you are used to separating the two languages. It is a good idea to start by using basic HTML in your JSX. Open the `App.js` file and remove all unnecessary HTML code as shown:

src/App.js

```
import React, { Component } from 'react';
import './App.css';

class App extends Component {
  render() {
    return (
      <div className="App">
        <h2>Welcome to the Road to learn React</h2>
      </div>
    );
  }
}

export default App;
```

Now, you only return HTML in your `render()` method without any JavaScript. Let's define the "Welcome to the Road to learn React" as a variable. A variable is set in JSX by curly braces.

src/App.js

```
import React, { Component } from 'react';
import './App.css';

class App extends Component {
  render() {
    var helloWorld = 'Welcome to the Road to learn React';
    return (
      <div className="App">
        <h2>{helloWorld}</h2>
      </div>
    );
  }
}

export default App;
```

Start your application on the command line with `npm start` to verify the changes you've made.

You might have noticed the `className` attribute. It reflects the standard `class` attribute in HTML. JSX had replaced a handful of internal HTML attributes, but you can find all the supported HTML attributes in React's documentation[54], which all follow the camelCase convention. On your way to learn React, expect to run across more JSX specific attributes.

Exercises:

- Confirm your source code for the last section[55]
  - Confirm the changes from the last section[56]
- Define more variables and render them in JSX
  - Use a complex object to represent a user with a first name and last name
  - Render the user properties in JSX
- Read about JSX[57]
- Read about React components, elements and instances[58]

[54] https://reactjs.org/docs/dom-elements.html#all-supported-html-attributes
[55] http://bit.ly/2H8H14h
[56] http://bit.ly/2H9KwHA
[57] https://reactjs.org/docs/introducing-jsx.html
[58] https://reactjs.org/blog/2015/12/18/react-components-elements-and-instances.html

ES6 const and let

Notice that we declared the variable `helloWorld` with a `var` statement. JavaScript ES6 comes with two more ways to declare variables: `const` and `let`. In JavaScript ES6, you will rarely find `var` anymore. A variable declared with `const` cannot be re-assigned or re-declared, and cannot be changed or modified. Once the variable is assigned, you cannot change it:

Code Playground

```
// not allowed
const helloWorld = 'Welcome to the Road to learn React';
helloWorld = 'Bye Bye React';
```

Conversely, a variable declared with `let` can be modified:

Code Playground

```
// allowed
let helloWorld = 'Welcome to the Road to learn React';
helloWorld = 'Bye Bye React';
```

TIP: Declare variables with `let` if you think you'll want to re-assign it later on.

Note that a variable declared directly with `const` cannot be modified. However, when the variable is an array or object, the values it holds can get updated through indirect means:

Code Playground

```
// allowed
const helloWorld = {
  text: 'Welcome to the Road to learn React'
};
helloWorld.text = 'Bye Bye React';
```

There are varying opinions about when to use *const* and when to use *let*. I would recommend using `const` whenever possible to show the intent of keeping your data structures immutable, so you only have to worry about the values contained in objects and arrays. Immutability is embraced in the React ecosystem, so `const` should be your default choice when you define a variable, though it's not really about immutability, but about assigning variables only once. It shows the intent of not changing (re-assigning) the variable even though its content can be changed.

src/App.js

```
import React, { Component } from 'react';
import './App.css';

class App extends Component {
  render() {
    const helloWorld = 'Welcome to the Road to learn React';
    return (
      <div className="App">
        <h2>{helloWorld}</h2>
      </div>
    );
  }
}

export default App;
```

In your application, we will use `const` and `let` over `var` for the rest of the book.

Exercises:

- Confirm your source code for the last section[59]
  - Confirm the changes from the last section[60]
- Read about ES6 const[61]
- Read about ES6 let[62]
- Gain an understanding of immutable data structures:
  - Why do they make sense in programming?
  - Why are they embraced in React and its ecosystem?

[59] http://bit.ly/2H9AqH2
[60] http://bit.ly/2H61Vkw
[61] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const
[62] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let

ReactDOM

The App component is located in your entry point to the React world: the *src/index.js* file.

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
import './index.css';

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

`ReactDOM.render()` uses a DOM node in your HTML to replace it with JSX. It's a way to integrate React in any foreign application easily, and you can use `ReactDOM.render()` multiple times across your application. You can use it to bootstrap simple JSX syntax, a React component, multiple React components, or an entire application. In a plain React application, you would only use it once to bootstrap the component tree.

`ReactDOM.render()` expects two arguments. The first argument is for rendering the JSX. The second argument specifies the place where the React application hooks into your HTML. It expects an element with an `id='root'`, found in the *public/index.html* file.

Code Playground

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

During implementation, `ReactDOM.render()` takes your App component, though it can also pass simple JSX. It doesn't require a component instance.

Exercises:

- Open the *public/index.html* to see where the React application hooks into your HTML
- Read about rendering elements in React[63]

[63] https://reactjs.org/docs/rendering-elements.html

Hot Module Replacement

Hot Module Replacement can be used in the *src/index.js* file to improve your experience as a developer. By default, *create-react-app* will cause the browser to refresh the page whenever its source code is modified. Try it by changing the `helloWorld` variable in your *src/App.js* file, which should cause the browser to refresh the page. There is a better way of handling source code changes during development, however.

Hot Module Replacement (HMR) is a tool for reloading your application in the browser without the page refresh. You can activate it in *create-react-app* by adding the following configuration to your *src/index.js* file:

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';

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

if (module.hot) {
  module.hot.accept();
}
```

Again, change the `helloWorld` variable in your *src/App.js* file. The browser shouldn't refresh, but the application will reload and show the correct output. HMR comes with multiple advantages:

Imagine you are debugging your code with `console.log()` statements. These statements will stay in your developer console, even though you changed your code, because the browser doesn't refresh the page anymore. In a growing application, page refreshes delay productivity; HMR removes this obstacle by eliminating the incremental time loss it takes for a browser to reload.

The most useful benefit of HMR is that you can keep the application state after the application reloads. For instance, assume you have a dialog or wizard in your application with multiple steps, and you are on step 3. Without HMR, you make changes to the source code and your browser refreshes the page. You would then have to open the dialog again and navigate from step 1 to step 3 each time. With HMR your dialog stays open at step 3, so you can debug from the exact point you're working on. With the time saved from page loads, this makes HMR an invaluable tool for React developers.

Exercises:

- Confirm your source code for the last section[64]
  - Confirm the changes from the last section[65]
- Change your *src/App.js* source code a few times to see HMR in action
- Watch the first 10 minutes of Live React: Hot Reloading with Time Travel[66] by Dan Abramov

[64] http://bit.ly/2H6pn1j
[65] http://bit.ly/2H8C6jW
[66] https://www.youtube.com/watch?v=xsSnOQynTHs

Complex JavaScript in JSX

So far, you have rendered a few primitive variables in your JSX. Now, we will render a list of items. The list will contain sample data in the beginning, but later we will learn how to fetch the data from an external API.

First you have to define the list of items:

src/App.js

```
import React, { Component } from 'react';
import './App.css';

const list = [
  {
    title: 'React',
    url: 'https://reactjs.org/',
    author: 'Jordan Walke',
    num_comments: 3,
    points: 4,
    objectID: 0,
  },
  {
    title: 'Redux',
    url: 'https://redux.js.org/',
    author: 'Dan Abramov, Andrew Clark',
    num_comments: 2,
    points: 5,
    objectID: 1,
  },
];

class App extends Component {
  ...
}
```

The sample data represents information we will fetch from an API later on. Items in this list each have a title, a url, and an author, as well an identifier, points (which indicate how popular an article is), and a count of comments.

Now you can use the built-in JavaScript map functionality[67] in JSX, which iterates over a list of items to display them according to specific attributes. Again, we use curly braces to encapsulate the JavaScript expression in JSX:

[67] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map

src/App.js

```
class App extends Component {
  render() {
    return (
      <div className="App">
        {list.map(function(item) {
          return <div>{item.title}</div>;
        })}
      </div>
    );
  }
}

export default App;
```

Using JavaScript alongside HTML in JSX is very powerful. For a different task you may have used `map` to convert one list of items to another. This time, we used `map` to convert a list of items to HTML elements.

Code Playground

```
const array = [1, 4, 9, 16];

// pass a function to map
const newArray = array.map(function (x) { return x * 2; });

console.log(newArray);
// expected output: Array [2, 8, 18, 32]
```

So far, only the `title` is displayed for each item. Let's experiment with more of the item's properties:

src/App.js

```
class App extends Component {
  render() {
    return (
      <div className="App">
        {list.map(function(item) {
          return (
            <div>
              <span>
                <a href={item.url}>{item.title}</a>
              </span>
```

```
              <span>{item.author}</span>
              <span>{item.num_comments}</span>
              <span>{item.points}</span>
            </div>
          );
        })}
      </div>
    );
  }
}

export default App;
```

Note how the `map` function is inlined in your JSX. Each item property is displayed with a `<span>` tag, and the url property of the item is in the `href` attribute of the anchor tag.

React will display each item, but you can still do more to help React embrace its full potential. By assigning a key attribute to each list element, React can identify modified items when the list changes. These sample list items come with an identifier:

src/App.js

```
{list.map(function(item) {
  return (
    <div key={item.objectID}>
      <span>
        <a href={item.url}>{item.title}</a>
      </span>
      <span>{item.author}</span>
      <span>{item.num_comments}</span>
      <span>{item.points}</span>
    </div>
  );
})}
```

Make sure that the key attribute is a stable identifier. Avoid using the index of the item in the array, because the array index is not stable. If the list changes its order, for example, React will not be able to identify the items properly.

src/App.js

```
// don't do this
{list.map(function(item, key) {
  return (
    <div key={key}>
      ...
    </div>
  );
})}
```

Start your app in a browser, and you should see both items of the list displayed.

Exercises:

- Confirm your source code for the last section[68]
  - Confirm the changes from the last section[69]
- Read about React lists and keys[70]
- Recap the standard built-in array functionalities in JavaScript[71] and discover a couple of use cases[72]
- Use more JavaScript expressions on your own in JSX

[68] http://bit.ly/2H7jMHT
[69] http://bit.ly/2H6LKnb
[70] https://reactjs.org/docs/lists-and-keys.html
[71] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/
[72] https://www.robinwieruch.de/javascript-map-array/

ES6 Arrow Functions

JavaScript ES6 introduced arrow functions expressions, which are shorter than a function expressions.

Code Playground

```
// function declaration
function () { ... }

// arrow function declaration
() => { ... }
```

You can remove the parentheses in an arrow function expression if it only has one argument, but you have to keep the parentheses if it gets multiple arguments:

Code Playground

```
// allowed
item => { ... }

// allowed
(item) => { ... }

// not allowed
item, key => { ... }

// allowed
(item, key) => { ... }
```

You can also write `map` functions more concisely with an ES6 arrow function:

src/App.js

```
{list.map(item => {
  return (
    <div key={item.objectID}>
      <span>
        <a href={item.url}>{item.title}</a>
      </span>
      <span>{item.author}</span>
      <span>{item.num_comments}</span>
      <span>{item.points}</span>
    </div>
```

```
  );
})}
```

You can remove the *block body*, the curly braces, with the ES6 arrow function. In a *concise body*, an implicit return is attached; thus, you can remove the `return` statement. This will happen often in this book, so be sure to understand the difference between a block body and a concise body when using arrow functions.

src/App.js

```
{list.map(item =>
  <div key={item.objectID}>
    <span>
      <a href={item.url}>{item.title}</a>
    </span>
    <span>{item.author}</span>
    <span>{item.num_comments}</span>
    <span>{item.points}</span>
  </div>
)}
```

Your JSX should look more concise and readable now, as it omits the `function` statement, the curly braces, and the return statement.

Exercises:

- Confirm your source code for the last section[73]
  - Confirm the changes from the last section[74]
- Read about ES6 arrow functions[75]

[73] http://bit.ly/2H6Mfh3
[74] http://bit.ly/2H92076
[75] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/Arrow_functions

ES6 Classes

JavaScript ES6 introduced classes, which are commonly used in object-oriented programming languages. JavaScript, always flexible in its programming paradigms, allows functional programming and object-oriented programming to work side-by-side.

While React embraces functional programming, e.g. immutable data structures and function compositions, classes are used to declare ES6 class components. React mixes the good parts of both programming paradigms.

Consider the following Developer class to examine a JavaScript ES6 class without a component.

Code Playground

```
class Developer {
  constructor(firstname, lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
  }

  getName() {
    return this.firstname + ' ' + this.lastname;
  }
}
```

A class has a constructor to make it instantiable. The constructor takes arguments and assigns them to the class instance. A class can also define functions. Because the function is associated with a class, it is called a method, or a class method.

The Developer class is the only class declaration we use here, as you can create multiple instances of a class by invoking it. It is similar to the ES6 class component, which has a declaration, but you have to use it somewhere else to instantiate it:

Code Playground

```
const robin = new Developer('Robin', 'Wieruch');
console.log(robin.getName());
// output: Robin Wieruch
```

React uses JavaScript ES6 classes for ES6 class components, which you have already used at least once so far:

src/App.js

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

...

class App extends Component {
  render() {
    ...
  }
}
```

When you declare the App component it extends from another component. In object-oriented programming, the term "extends" refers to the principle of inheritance, which means that functionality can be passed from one class to another. The App class extends from the Component class, meaning it inherits functionality from the Component class. The Component class is used to extend a basic ES6 class to a ES6 component class. It has all the functionalities that a component in React needs. The render method is one function you have already used. You will learn about other component class methods as we move along.

The `Component` class encapsulates all the implementation details of a React component, which allows developers to use classes as components in React.

Methods exposed by a React `Component` are its public interface. One of these methods must be overridden, while the others don't need to be overridden. You will learn about these when we discuss lifecycle methods later. The `render()` method has to be overridden, because it defines the output of a React `Component`, so it must be defined. These are the basics of JavaScript ES6 classes, and how they are used in React to extend them to components.

Exercises:

- Read about JavaScript fundamentals before learning React[76]
- Read about ES6 classes[77]

[76] https://www.robinwieruch.de/javascript-fundamentals-react-requirements/
[77] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes

Congratulations, you have learned to bootstrap your first React application! Let's recap:

- **React**
  - Create-react-app bootstraps a React application
  - JSX mixes up HTML and JavaScript to define the output of React components in their render methods
  - Components, instances, and elements are different items in React
  - `ReactDOM.render()` is an entry point for a React application to hook React into the DOM
  - Built-in JavaScript functionalities can be used in JSX
  - Map can be used to render a list of items as HTML elements
- **ES6**
  - Variable declarations with `const` and `let` can be used for specific use cases
  - Use const over let in React applications
  - Arrow functions can be used to keep your functions concise
  - Classes are used to define components in React by extending them

Now that you've completed the first chapter, it's advisable to experiment with the source code you have written so far and see what changes you can make on your own. You can find the source code in the official repository[78].

[78] http://bit.ly/2H6Mfh3

Basics in React

This chapter will guide you through the basics of React. It covers state and interactions in components as we move past static components. We will also cover the different ways to declare a component, and how to keep components composable and reusable.

Local Component State

Local component state, also known as internal component state, allows you to save, modify, and delete properties stored in your component. The ES6 class component then uses a constructor to initialize local component state. The constructor is called only once, when the component initializes:

Let's introduce a class constructor.

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);
  }

  ...

}
```

The App component is a subclass of `Component`, so the `extends Component` is in the App component declaration.

It is mandatory to call `super(props);`. It sets `this.props` in your constructor in case you want to access them there. They would be `undefined` when accessing `this.props` in your constructor otherwise. In this case, the initial state of the component should be the sample list of items:

src/App.js

```
const list = [
  {
    title: 'React',
    url: 'https://reactjs.org/',
    author: 'Jordan Walke',
    num_comments: 3,
    points: 4,
    objectID: 0,
  },
  ...
];

class App extends Component {

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

```
    this.state = {
      list: list,
    };
  }

  ...

}
```

The state is bound to the class using the `this` object, so you can access the local state of the whole component. For instance, it can be used in the `render()` method. Previously you have mapped a static list of items in your `render()` method that was defined outside of your component. Now you are about to use the list from your local state in your component.

src/App.js

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        {this.state.list.map(item =>
          <div key={item.objectID}>
            <span>
              <a href={item.url}>{item.title}</a>
            </span>
            <span>{item.author}</span>
            <span>{item.num_comments}</span>
            <span>{item.points}</span>
          </div>
        )}
      </div>
    );
  }
}
```

The list is part of the component now, in the local component state. You could add, change, or remove items from your list. Every time you change your component state, the `render()` method of your component will run again. That's how you can change your local component state and see the component re-render the correct data from the local state.

Be careful not to mutate the state directly. Instead, you should use a method called `setState()` to modify your states. We will cover these concepts in more depth next chapter.

Exercises:

- Confirm your source code for the last section[79]
  - Confirm the changes from the last section[80]
- Experiment with the local state
  - Define more initial state in the constructor
  - Use and access the state in your `render()` method
- Read about the ES6 class constructor[81]

[79] http://bit.ly/2Had8Rh
[80] http://bit.ly/2HacLWT
[81] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes#Constructor

ES6 Object Initializer

In JavaScript ES6, you can use a shorthand property syntax to initialize your objects more concisely, like following object initialization:

Code Playground

```
const name = 'Robin';

const user = {
  name: name,
};
```

When the property name in your object is the same as your variable name, you can do the following:

Code Playground

```
const name = 'Robin';

const user = {
  name,
};
```

In your application, you can do the same. The list variable name and the state property name share the same name.

Code Playground

```
// ES5
this.state = {
  list: list,
};

// ES6
this.state = {
  list,
};
```

Shorthand method names are also useful. In JavaScript ES6, you can initialize methods in an object more concisely:

Code Playground

```
// ES5
var userService = {
  getUserName: function (user) {
    return user.firstname + ' ' + user.lastname;
  },
};

// ES6
const userService = {
  getUserName(user) {
    return user.firstname + ' ' + user.lastname;
  },
};
```

Finally, you are allowed to use computed property names in JavaScript ES6:

Code Playground

```
// ES5
var user = {
  name: 'Robin',
};

// ES6
const key = 'name';
const user = {
  [key]: 'Robin',
};
```

Later, you will be able to use computed property names to allocate values by key in an object dynamically, a handy way to generate lookup tables in JavaScript.

Exercises:

- Confirm your source code for the last section[82]
  - Confirm the changes from the last section[83]
- Experiment with ES6 object initializer
- Read about ES6 object initializer[84]

[82] http://bit.ly/2H9Ewif
[83] http://bit.ly/2HadAyX
[84] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Object_initializer

Unidirectional Data Flow

Now you have a local state in your App component, but haven't manipulated the state yet. The local state is static, so its component is as well. A good way to experience state manipulation is to engage in component interaction.

We will practice this concept by adding a button for each item in the displayed list. The button will read "Dismiss", as its purpose will be to remove an item from the list. In an email client, for example, it would be a useful way to mark some list items as 'read' while keeping the unread items separate.

src/App.js

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        {this.state.list.map(item =>
          <div key={item.objectID}>
            <span>
              <a href={item.url}>{item.title}</a>
            </span>
            <span>{item.author}</span>
            <span>{item.num_comments}</span>
            <span>{item.points}</span>
            <span>
              <button
                onClick={() => this.onDismiss(item.objectID)}
                type="button"
              >
                Dismiss
              </button>
            </span>
          </div>
        )}
      </div>
    );
  }
}
```

The `onDismiss()` class method is not defined yet. We will do it in a moment, but for now, let's focus on the `onClick` handler of the button element. As you can see, the `onDismiss()` method in the

`onClick` handler is enclosed by an arrow function. You can use it to peek at the `objectID` property of the `item` object and identify the item to be dismissed. An alternative way would be to define the function outside of the `onClick` handler and only pass the defined function to it. We will cover handlers more in-depth as we move along.

Note the use of multilines for the button element, and how elements with multiple attributes can get disorganized easily. The button element is used with multilines and indentations to keep it readable. While this practice isn't specific to React development, it is a programming style I recommend for cleanliness and your own peace of mind.

Now we will implement the `onDismiss()` functionality. It takes an id to identify the item to dismiss. The function is bound to the class and thus becomes a class method. That's why you access it with `this.onDismiss()` and not `onDismiss()`. The `this` object is your class instance. In order to define the `onDismiss()` as class method, you have to bind it in the constructor:

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      list,
    };

    this.onDismiss = this.onDismiss.bind(this);
  }

  render() {
    ...
  }
}
```

In the next step, we define its functionality, the business logic, in the class:

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      list,
    };

    this.onDismiss = this.onDismiss.bind(this);
  }

  onDismiss(id) {
    ...
  }

  render() {
    ...
  }
}
```

Now we can define what happens inside a class method. Remember, the objective is to remove the item identified by the id from the list and store an updated list to the local state. The updated list will be used in the re-running `render()` method to display it, where the removed item should no longer appear.

You can remove an item from a list using JavaScript's built-in filter[85] functionality, which takes a function as input. The function has access to each value in the list because it iterates over each item, so you can evaluate them based on certain conditions.

Code Playground

```
const words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];

const filteredWords = words.filter(function (word) { return word.length > 6; });

console.log(filteredWords);
// expected output: Array ["exuberant", "destruction", "present"]
```

The function returns a new list instead of mutating the old one, and it supports the React convention of using immutable data structures.

[85] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter

src/App.js

```
onDismiss(id) {
  const updatedList = this.state.list.filter(function isNotId(item) {
    return item.objectID !== id;
  });
}
```

In the next step, we extract the function and pass it to the filter function:

src/App.js

```
onDismiss(id) {
  function isNotId(item) {
    return item.objectID !== id;
  }

  const updatedList = this.state.list.filter(isNotId);
}
```

Remember: You can filter more efficiently using a JavaScript ES6 arrow function.

src/App.js

```
onDismiss(id) {
  const isNotId = item => item.objectID !== id;
  const updatedList = this.state.list.filter(isNotId);
}
```

You could even inline it again like you did in the `onClick` handler of the button, though it might get less readable:

src/App.js

```
onDismiss(id) {
  const updatedList = this.state.list.filter(item => item.objectID !== id);
}
```

The list removes the clicked item now, but the state hasn't updated yet. Use the `setState()` class method to update the list in the local component state:

src/App.js

```
onDismiss(id) {
  const isNotId = item => item.objectID !== id;
  const updatedList = this.state.list.filter(isNotId);
  this.setState({ list: updatedList });
}
```

Run the application again and try the "Dismiss" button. What you experienced is the **unidirectional data flow** of React. An action is triggered in the view layer with `onClick()`, a function or class method modifies the local component state, and then the `render()` method of the component runs again to update the view.

Exercises:

- Confirm your source code for the last section[86]
  - Confirm the changes from the last section[87]
- Read about the state and lifecycle in React[88]

[86] http://bit.ly/2H928n6
[87] http://bit.ly/2Ha1s0S
[88] https://reactjs.org/docs/state-and-lifecycle.html

Bindings

It is important to learn about bindings in JavaScript classes when using React ES6 class components. In the previous chapter, you have bound your class method `onDismiss()` in the constructor.

src/App.js

```
class App extends Component {
  constructor(props) {
    super(props);

    this.state = {
      list,
    };

    this.onDismiss = this.onDismiss.bind(this);
  }

  ...
}
```

The binding step is necessary because class methods don't automatically bind `this` to the class instance. Let's demonstrate it with the help of the following ES6 class component:

Code Playground

```
class ExplainBindingsComponent extends Component {
  onClickMe() {
    console.log(this);
  }

  render() {
    return (
      <button
        onClick={this.onClickMe}
        type="button"
      >
        Click Me
      </button>
    );
  }
}
```

The component renders just fine, but when you click the button, you see `undefined` in your developer console log. This is one of the main sources of bugs developers encounter in React. If you want to access `this.state` in your class method, it cannot be retrieved because `this` is `undefined`. To make `this` accessible in your class methods, you have to bind the class methods to `this`.

In the following class component the class method is properly bound in the class constructor:

Code Playground

```
class ExplainBindingsComponent extends Component {
  constructor() {
    super();

    this.onClickMe = this.onClickMe.bind(this);
  }

  onClickMe() {
    console.log(this);
  }

  render() {
    return (
      <button
        onClick={this.onClickMe}
        type="button"
      >
        Click Me
      </button>
    );
  }
}
```

Class method binding can happen somewhere else too. For instance, it can happen in the `render()` class method:

Code Playground

```
class ExplainBindingsComponent extends Component {
  onClickMe() {
    console.log(this);
  }

  render() {
    return (
      <button
```

```
        onClick={this.onClickMe.bind(this)}
        type="button"
      >
        Click Me
      </button>
    );
  }
}
```

Avoid this practice, however, because it binds the class method every time the `render()` method runs, meaning every time the component updates, which will hurt your application's performance eventually. Binding the class method in the constructor need only be done once, when the component is instantiated.

Some developers will define the business logic of their class methods in the constructor:

Code Playground

```
class ExplainBindingsComponent extends Component {
  constructor() {
    super();

    this.onClickMe = () => {
      console.log(this);
    }
  }

  render() {
    return (
      <button
        onClick={this.onClickMe}
        type="button"
      >
        Click Me
      </button>
    );
  }
}
```

Avoid this approach as well, as it will clutter your constructor over time. The constructor is only there to instantiate your class with all its properties, so the business logic of class methods should be defined outside the constructor.

Code Playground

```
class ExplainBindingsComponent extends Component {
  constructor() {
    super();

    this.doSomething = this.doSomething.bind(this);
    this.doSomethingElse = this.doSomethingElse.bind(this);
  }

  doSomething() {
    // do something
  }

  doSomethingElse() {
    // do something else
  }

  ...
}
```

Class methods can be auto-bound using JavaScript ES6 arrow functions:

Code Playground

```
class ExplainBindingsComponent extends Component {
  onClickMe = () => {
    console.log(this);
  }

  render() {
    return (
      <button
        onClick={this.onClickMe}
        type="button"
      >
        Click Me
      </button>
    );
  }
}
```

Use this method if the repetitive binding in the constructor annoys you. The official React documentation sticks to the class method bindings in the constructor, so this book will stick with those as well.

Exercises:

- Try different approaches of bindings and console log the `this` object
- Learn more about an alternative React component syntax[89]

[89] https://github.com/the-road-to-learn-react/react-alternative-class-component-syntax

Event Handler

Now we'll cover event handlers in elements. In your application, you are using the following button element to dismiss an item from the list.

src/App.js

```
...

<button
  onClick={() => this.onDismiss(item.objectID)}
  type="button"
>
  Dismiss
</button>

...
```

This function is already complex, because it passes a value to the class method and has to wrap it in another (arrow) function. Essentially, it has to be a function that is passed to the event handler. The following code wouldn't work, because the class method would be executed immediately when you open the application in the browser:

src/App.js

```
...

<button
  onClick={this.onDismiss(item.objectID)}
  type="button"
>
  Dismiss
</button>

...
```

When using `onClick={doSomething()}`, the `doSomething()` function executes immediately when the application is opened in a browser. The expression in the handler is evaluated. Since the returned value of the function isn't a function anymore, nothing would happen when you click the button. But using `onClick={doSomething}` where `doSomething` is a function, it would only be executed if the button is clicked. The same rules apply for the `onDismiss()` class method.

However, using `onClick={this.onDismiss}` wouldn't suffice, because the `item.objectID` property needs to be passed to the class method to identify the item that should be dismissed. We wrap it into another function to sneak in the property. This concept is called higher-order functions in JavaScript, which we will cover briefly later.

src/App.js

```
...

<button
  onClick={() => this.onDismiss(item.objectID)}
  type="button"
>
  Dismiss
</button>

...
```

We can also define wrapping function outside the method, to pass only the defined function to the handler. Since it needs access to the individual item, it has to live inside the map function block.

src/App.js

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        {this.state.list.map(item => {
          const onHandleDismiss = () =>
            this.onDismiss(item.objectID);

          return (
            <div key={item.objectID}>
              <span>
                <a href={item.url}>{item.title}</a>
              </span>
              <span>{item.author}</span>
              <span>{item.num_comments}</span>
              <span>{item.points}</span>
              <span>
                <button
                  onClick={onHandleDismiss}
                  type="button"
                >
                  Dismiss
                </button>
```

```
              </span>
            </div>
          );
        }
        )}
      </div>
    );
  }
}
```

A function has to be passed to the element's handler. As an example, try this code instead:

src/App.js

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        {this.state.list.map(item =>
            ...
            <span>
              <button
                onClick={console.log(item.objectID)}
                type="button"
              >
                Dismiss
              </button>
            </span>
          </div>
        )}
      </div>
    );
  }
}
```

This method will run when you open the application in the browser, but not when you click the button. The following code would only run when you click the button, a function that is executed when you trigger the handler:

src/App.js

```
...

<button
  onClick={function () {
    console.log(item.objectID)
  }}
  type="button"
>
  Dismiss
</button>

...
```

Remember: You can transform functions into a JavaScript ES6 arrow function, just as we did with the `onDismiss()` class method:

src/App.js

```
...

<button
  onClick={() => console.log(item.objectID)}
  type="button"
>
  Dismiss
</button>

...
```

Newcomers to React often have difficulty using functions in event handlers, so don't get discouraged if you have trouble on the first pass. You should end up with an inlined JavaScript ES6 arrow function with access to the `objectID` property of the `item` object:

src/App.js

```
class App extends Component {
  ...

  render() {
    return (
      <div className="App">
        {this.state.list.map(item =>
          <div key={item.objectID}>
            ...
            <span>
              <button
                onClick={() => this.onDismiss(item.objectID)}
                type="button"
              >
                Dismiss
              </button>
            </span>
          </div>
        )}
      </div>
    );
  }
}
```

Using arrow functions in event handlers directly impacts your application's performance. For instance, the `onClick` handler for the `onDismiss()` method wraps the method in another arrow function to pass the item identifier. Every time the `render()` method runs, the handler instantiates the higher-order arrow function. It can have an impact on your application performance, but in most cases you won't notice. If you have a huge table of data with 1000 items and each row or column has an arrow function in an event handler, it is worth thinking about the performance implications, so you could implement a dedicated Button component to bind the method in the constructor. Before that, though, it is premature optimization, and it is more prudent to learn the basics of React before thinking about optimization.

Exercises:

- Try the different approaches of using functions in the `onClick` handler of your button.

Interactions with Forms and Events

We'll add another interaction to see forms and events in React, a search functionality where the input in the search field temporarily filters a list based on the title property of an item.

In the first step, we define a form with an input field in JSX:

src/App.js

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        <form>
          <input type="text" />
        </form>
        {this.state.list.map(item =>
          ...
        )}
      </div>
    );
  }
}
```

In the following scenario you will type into the input field and filter the list temporarily by the search term that is used in the input field. To filter the list based on the value of the input field, we store the value of the input field in the local state. We use **synthetic events** in React to access a value in an event payload.

Let's define a `onChange` handler for the input field:

src/App.js

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        <form>
          <input
```

```
            type="text"
            onChange={this.onSearchChange}
          />
        </form>
        ...
      </div>
    );
  }
}
```

The function is bound to the component, so it is a class method again. You just need to bind and define the method:

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      list,
    };

    this.onSearchChange = this.onSearchChange.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
  }

  onSearchChange() {
    ...
  }

  ...
}
```

When using a handler in your element, you get access to the synthetic React event in your callback function's signature.

src/App.js

```
class App extends Component {

  ...

  onSearchChange(event) {
    ...
  }

  ...
}
```

The event has the value of the input field in its target object, so you can update the local state with a search term using `this.setState()`.

src/App.js

```
class App extends Component {

  ...

  onSearchChange(event) {
    this.setState({ searchTerm: event.target.value });
  }

  ...
}
```

Don't forget to define the initial state for the `searchTerm` property in the constructor. The input field should be empty in the beginning, so its value is an empty string.

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      list,
      searchTerm: '',
    };
```

```
    this.onSearchChange = this.onSearchChange.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
  }

  ...
}
```

We store the input value to the local state every time the value in the input field changes.

We can assume that when we update `searchTerm` with `this.setState()`, the list also needs to be passed to preserve it. React's `this.setState()` is a shallow merge, however, so it preserves the sibling properties in the state object when it updates a property. The list state, though you have already dismissed an item from it, stays the same when updating the `searchTerm` property.

Returning to the application, we see the list isn't filtered yet, based on the input field value stored in the local state. We need to filter the list temporarily based on the `searchTerm`, and we have everything we need to perform this operation. In the `render()` method, before mapping over the list, we apply a filter to it. The filter will only evaluate if the `searchTerm` matches the title property of the item. We've already used the built-in JavaScript filter functionality, so let's use it again to sneak in the filter function before the map function. The filter function returns a new array, so the map function can be used on it.

src/App.js

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        <form>
          <input
            type="text"
            onChange={this.onSearchChange}
          />
        </form>
        {this.state.list.filter(...).map(item =>
          ...
        )}
      </div>
    );
  }
}
```

Let's approach the filter function in a different way this time. We want to define the filter argument, which is the function passed to the filter function outside the ES6 class component. We don't have access to the state of the component, so we have no access to the `searchTerm` property to evaluate the filter condition. This means we'll need to pass the `searchTerm` to the filter function, returning a new function to evaluate the condition. This is called a higher-order function.

It makes sense to know about higher-order functions, because React deals with a concept called higher-order components. You will get to know the concept later in the book. Now again, let's focus on the filter functionality.

First, you have to define the higher-order function outside of your App component.

src/App.js

```
function isSearched(searchTerm) {
  return function (item) {
    // some condition which returns true or false
  }
}

class App extends Component {

  ...

}
```

The function takes the `searchTerm` and returns another function, because the filter function only takes that type as its input. The returned function has access to the item object, because it is the one passed to the filter function.

It will also be used to filter the list based on the condition defined in the function, so let's define the condition:

src/App.js

```
function isSearched(searchTerm) {
  return function (item) {
    return item.title.toLowerCase().includes(searchTerm.toLowerCase());
  }
}

class App extends Component {

  ...

}
```

The condition matches the incoming `searchTerm` pattern with the title property of the item from your list. You can do that with the built-in `includes` JavaScript functionality. When the pattern matches, it returns true and the item stays in the list; when the pattern doesn't match, the item is removed from the list. Don't forget to match the capitalization on both strings to the letter, as there will be mismatches between the search term 'redux' and an item title 'Redux'. Since we are working on an immutable list and return a new list by using the filter function, the original list in the local state isn't modified at all.

We cheated a bit using JavaScript ES7 features, but these aren't present in ES5. For ES5, use the `indexOf()` function to get the index of the item in the list instead. When the item is in the list, `indexOf()` will return its index in the array.

Code Playground

```
// ES5
string.indexOf(pattern) !== -1

// ES6
string.includes(pattern)
```

Another neat refactoring can be done with an ES6 arrow function again. It makes the function more concise:

Code Playground

```
// ES5
function isSearched(searchTerm) {
  return function (item) {
    return item.title.toLowerCase().indexOf(searchTerm.toLowerCase()) !== -1;
  }
}

// ES6
const isSearched = searchTerm => item =>
  item.title.toLowerCase().includes(searchTerm.toLowerCase());
```

The React ecosystem uses a lot of functional programming concepts, often using functions that return functions (the concept is called high-order functions) to pass information. JavaScript ES6 lets us express these even more concisely with arrow functions.

Last but not least, use the defined `isSearched()` function to filter lists. We pass it the `searchTerm` property from the local state, so that it returns the filter's input function and filters your list based on the filter condition. After that it maps over the filtered list to display an element for each list item.

src/App.js

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        <form>
          <input
            type="text"
            onChange={this.onSearchChange}
          />
        </form>
        {this.state.list.filter(isSearched(this.state.searchTerm)).map(item =>
          ...
        )}
      </div>
    );
  }
}
```

The search functionality should work now. Try it yourself in the browser.

Exercises:

- Confirm your source code for the last section[90]
  - Confirm the changes from the last section[91]
- Read about React events[92]
- Read about higher-order functions[93]

[90] http://bit.ly/2HaaV8y
[91] http://bit.ly/2HaB8DB
[92] https://reactjs.org/docs/handling-events.html
[93] https://en.wikipedia.org/wiki/Higher-order_function

ES6 Destructuring

Destructuring in JavaScript ES6 provides easier access to properties in objects and arrays. Compare the following snippet in JavaScript ES5 and ES6:

Code Playground

```
const user = {
  firstname: 'Robin',
  lastname: 'Wieruch',
};

// ES5
var firstname = user.firstname;
var lastname = user.lastname;

console.log(firstname + ' ' + lastname);
// output: Robin Wieruch

// ES6
const { firstname, lastname } = user;

console.log(firstname + ' ' + lastname);
// output: Robin Wieruch
```

While we add an extra line each time we access an object property in JavaScript ES5, it takes just one line in JavaScript ES6. For readability, use multilines when you destructure an object into multiple properties.

Code Playground

```
const {
  firstname,
  lastname
} = user;
```

The same concept applies to arrays. You can destructure them, too, again using multilines to keep your code scannable and readable.

Code Playground

```
const users = ['Robin', 'Andrew', 'Dan'];
const [
  userOne,
  userTwo,
  userThree
] = users;

console.log(userOne, userTwo, userThree);
// output: Robin Andrew Dan
```

Note that the local state object in the App component can get destructured the same way. You can shorten the filter and map line of code.

src/App.js

```
  render() {
    const { searchTerm, list } = this.state;
    return (
      <div className="App">
        ...
        {list.filter(isSearched(searchTerm)).map(item =>
          ...
        )}
      </div>
    );
```

You can do it the ES5 or ES6 way:

Code Playground

```
// ES5
var searchTerm = this.state.searchTerm;
var list = this.state.list;

// ES6
const { searchTerm, list } = this.state;
```

But since the book uses JavaScript ES6 most of the time, you should stick to it.

Exercises:

- Confirm your source code for the last section[94]
  - Confirm the changes from the last section[95]
- Read about ES6 destructuring[96]

[94] http://bit.ly/2Hae4oL
[95] http://bit.ly/2H9EDub
[96] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

Controlled Components

We covered unidirectional data flows before, and the same law applies for the input field, which updates the local state with the `searchTerm` to filter the list. When the state changes, the `render()` method runs again and uses the recent `searchTerm` from the local state to apply the filter condition.

But didn't we forget something in the input element? An HTML input tag comes with a `value` attribute. The value attribute usually contains the value shown in the input field. In this case, that is the `searchTerm` property. Form elements such as `<input>`, `<textarea>`, and `<select>` hold their own state in plain HTML. They modify the value internally once someone changes it from the outside. In React, that's called an **uncontrolled component**, because it handles its own state. We want to make sure those elements are **controlled components** instead.

To do this, we set the value attribute of the input field, which is already saved in the `searchTerm` state property, so we can access it from there:

src/App.js

```
class App extends Component {

  ...

  render() {
    const { searchTerm, list } = this.state;
    return (
      <div className="App">
        <form>
          <input
            type="text"
            value={searchTerm}
            onChange={this.onSearchChange}
          />
        </form>
        ...
      </div>
    );
  }
}
```

The unidirectional data flow loop for the input field is self-contained, and the local component state is the single source of truth for the input field.

Local state management and unidirectional data flow might be new to you, but once you adjust to it, it will likely become your natural flow of React implementation. React brings novel patterns

with unidirectional data flow, which have been adopted by several frameworks and libraries which create single page applications.

Exercises:

- Confirm your source code for the last section[97]
  - Confirm the changes from the last section[98]
- Read about React forms[99]
- Learn more about different controlled components[100]

[97] http://bit.ly/2Habuzc
[98] http://bit.ly/2HaRQ5P
[99] https://reactjs.org/docs/forms.html
[100] https://github.com/the-road-to-learn-react/react-controlled-components-examples

Split Up Components

Now we have one large App component that keeps growing and may eventually become too complex to manage efficiently. We need to split it into smaller, more manageable parts by creating separate components for search input and the items list.

src/App.js

```
class App extends Component {

  ...

  render() {
    const { searchTerm, list } = this.state;
    return (
      <div className="App">
        <Search />
        <Table />
      </div>
    );
  }
}
```

We pass the components properties that they can use themselves. The App component needs to pass the properties managed in the local state and its class methods.

src/App.js

```
class App extends Component {

  ...

  render() {
    const { searchTerm, list } = this.state;
    return (
      <div className="App">
        <Search
          value={searchTerm}
          onChange={this.onSearchChange}
        />
        <Table
          list={list}
          pattern={searchTerm}
          onDismiss={this.onDismiss}
```

```
        />
      </div>
    );
  }
}
```

Now we define the components next to the App component, which will be done using JavaScript ES6 by using classes. They render the same elements as before.

The first one is the Search component:

src/App.js

```
class App extends Component {
  ...
}

class Search extends Component {
  render() {
    const { value, onChange } = this.props;
    return (
      <form>
        <input
          type="text"
          value={value}
          onChange={onChange}
        />
      </form>
    );
  }
}
```

The second one is the Table component.

src/App.js

```
...

class Table extends Component {
  render() {
    const { list, pattern, onDismiss } = this.props;
    return (
      <div>
        {list.filter(isSearched(pattern)).map(item =>
```

```
          <div key={item.objectID}>
            <span>
              <a href={item.url}>{item.title}</a>
            </span>
            <span>{item.author}</span>
            <span>{item.num_comments}</span>
            <span>{item.points}</span>
            <span>
              <button
                onClick={() => onDismiss(item.objectID)}
                type="button"
              >
                Dismiss
              </button>
            </span>
          </div>
        )}
      </div>
    );
  }
}
```

Now you have three ES6 class components. Notice the `props` object is accessible via the class instance by using `this`. Props, short for properties, have all the values passed to the components when we used App component. That way, components can pass properties down the component tree.

By extracting these components from the App component, they become reusable. Since components get their values using the `props` object, you can pass different props to your components every time you use them somewhere else.

Exercises:

- Confirm your source code for the last section[101]
  - Confirm the changes from the last section[102]
- Discover more components that can be split up like the Search and Table components, but wait until we've covered more of its concepts before you implement any of them.

[101] http://bit.ly/2H84S47
[102] http://bit.ly/2H62M4I

Composable Components

The `children` prop is used to pass elements to components from above, which are unknown to the component itself but make it possible to compose components together. We'll see how this looks when you pass a text string as a child to the Search component.

src/App.js

```
class App extends Component {

  ...

  render() {
    const { searchTerm, list } = this.state;
    return (
      <div className="App">
        <Search
          value={searchTerm}
          onChange={this.onSearchChange}
        >
          Search
        </Search>
        <Table
          list={list}
          pattern={searchTerm}
          onDismiss={this.onDismiss}
        />
      </div>
    );
  }
}
```

Now the Search component can destructure the `children` property from the `props` object, and specify where it should be displayed.

src/App.js

```
class Search extends Component {
  render() {
    const { value, onChange, children } = this.props;
    return (
      <form>
        {children} <input
          type="text"
          value={value}
          onChange={onChange}
        />
      </form>
    );
  }
}
```

The "Search" text should now be visible next to your input field. When you use the Search component elsewhere, you can use different entities, since it's not just text that can be passed as children. You can also pass an element, or element trees that can be encapsulated by components, as children. The children property makes it possible to weave components into each other.

Exercises:

- Confirm your source code for the last section[103]
  - Confirm the changes from the last section[104]
- Read about the composition model of React[105] and why/how to conduct component composition in React[106]

[103] http://bit.ly/2H82ZVg
[104] http://bit.ly/2Hb4FwY
[105] https://reactjs.org/docs/composition-vs-inheritance.html
[106] https://www.robinwieruch.de/react-component-composition/

Reusable Components

Reusable and composable components empower you to come up with capable component hierarchies, the foundation of React's view layer. The last sections mentioned reusability, and now we can see how reusing the Table and Search components works in our case. Even the App component is reusable, as it can be instantiated elsewhere as well.

Let's define one more reusable component, a Button component which we'll eventually reuse often:

src/App.js

```
class Button extends Component {
  render() {
    const {
      onClick,
      className,
      children,
    } = this.props;

    return (
      <button
        onClick={onClick}
        className={className}
        type="button"
      >
        {children}
      </button>
    );
  }
}
```

It might seem redundant to declare components like this, but it's not. We use a `Button` component instead of a `button` element, which spares only the `type="button"`. It might not seem like a huge win, but these measures are about long term, however. Imagine you have several buttons in your application, and you want to change an attribute, style, or behavior for just one. Without the component, you'd have to change (refactor) each one. The Button component ensures that the operation has a single source of truth, or one Button to refactor all the others at once.

Since you already have a button element, you can use the Button component instead. It omits the type attribute, because the Button component specifies it.

src/App.js

```
class Table extends Component {
  render() {
    const { list, pattern, onDismiss } = this.props;
    return (
      <div>
        {list.filter(isSearched(pattern)).map(item =>
          <div key={item.objectID}>
            <span>
              <a href={item.url}>{item.title}</a>
            </span>
            <span>{item.author}</span>
            <span>{item.num_comments}</span>
            <span>{item.points}</span>
            <span>
              <Button onClick={() => onDismiss(item.objectID)}>
                Dismiss
              </Button>
            </span>
          </div>
        )}
      </div>
    );
  }
}
```

The Button component expects a `className` property in the `props`. The `className` attribute is another React derivate for the HTML attribute class. We didn't pass any `className` when the Button was used, though. It should be more explicit in our Button component that the `className` is optional, so we'll assign a default value in the object destructuring.

src/App.js

```
class Button extends Component {
  render() {
    const {
      onClick,
      className = '',
      children,
    } = this.props;

    ...
```

```
  }
}
```

Now, whenever there is no `className` property specified in the Button component, the value will be an empty string instead of `undefined`.

Exercises:

- Confirm your source code for the last section[107]
  - Confirm the changes from the last section[108]
- Read about how to pass props in React[109]

[107] http://bit.ly/2HcAyp4
[108] http://bit.ly/2H6q3Up
[109] https://www.robinwieruch.de/react-pass-props-to-component/

Component Declarations

Now we have four ES6 class components, but the application can still be improved using functional stateless components as alternative for ES6 class components. Before you refactor your components, let's introduce the different types.

- **Functional Stateless Components** are functions that take input and return an output. The inputs are the props, and the output is a component instance in plain JSX. So far, it is quite similar to an ES6 class component. However, functional stateless components are functions (functional) and they have no local state (stateless). You cannot access or update the state with `this.state` or `this.setState()` because there is no `this` object. Additionally, they have no lifecycle methods except for the `render()` method which will be applied implicitly in functional stateless components. You didn't learn about lifecycle methods yet, but you already used two: `constructor()` and `render()`. The constructor runs only once in the lifetime of a component, whereas the `render()` class method runs once in the beginning and every time the component updates. Keep in mind that functional stateless components have no lifecycle methods, when we arrive at lifecycle methods chapter later.
- **ES6 Class Components** extend from the React component. The `extend` hooks all the lifecycle methods, available in the React component API, to the component. This is how we were able to use the `render()` class method. You can also store and manipulate state in ES6 class components using `this.state` and `this.setState()`.
- **React.createClass** was used in older versions of React, and is still used in JavaScript ES5 React applications. But Facebook declared it as deprecated[110] in favor of JavaScript ES6. They even added a deprecation warning in version 15.5[111], so we will not use it in the book.

When deciding when to use functional stateless components over ES6 class components, a good rule of thumb is to use functional stateless components when you don't need local state or component lifecycle methods. Usually, we implement components as functional stateless components, but once access to the state or lifecycle methods is required, we have to refactor it to an ES6 class component. We started the other way around in our application for the sake of learning.

Returning to the application, we see the App component uses local state, so it has to stay as an ES6 class component. The other three ES6 class components are stateless, so they don't need access to `this.state` or `this.setState()`, and they have no lifecycle methods. We're going to refactor the Search component to a stateless functional component. The Table and Button component refactoring will become your exercise.

[110] https://reactjs.org/blog/2015/03/10/react-v0.13.html
[111] https://reactjs.org/blog/2017/04/07/react-v15.5.0.html

src/App.js

```
function Search(props) {
  const { value, onChange, children } = props;
  return (
    <form>
      {children} <input
        type="text"
        value={value}
        onChange={onChange}
      />
    </form>
  );
}
```

The `props` are accessible in the function signature, and the return value is JSX; but we can do more with the code in a functional stateless component using ES6 destructuring. The best practice is to use it in the function signature to destructure the `props`:

src/App.js

```
function Search({ value, onChange, children }) {
  return (
    <form>
      {children} <input
        type="text"
        value={value}
        onChange={onChange}
      />
    </form>
  );
}
```

Remember that ES6 arrow functions allow you to keep your functions concise and let you remove the block body of the function. In a concise body, an implicit return is attached, letting us remove the `return` statement. Since the functional stateless component is a function, it can be made more concise as well:

src/App.js

```
const Search = ({ value, onChange, children }) =>
  <form>
    {children} <input
      type="text"
      value={value}
      onChange={onChange}
    />
  </form>
```

The last step is especially useful to enforce props as input and JSX as output. Still, you could *do something* in between by using a block body in your ES6 arrow function:

Code Playground

```
const Search = ({ value, onChange, children }) => {

  // do something

  return (
    <form>
      {children} <input
        type="text"
        value={value}
        onChange={onChange}
      />
    </form>
  );
}
```

Now you have one lightweight functional stateless component. When we need access to the local component state or lifecycle methods, we can refactor it to a ES6 class component. When using block bodies, programmers tend to make their functions more complex, but leaving the block body out lets you focus on the input and output. JavaScript ES6 in React components makes components more readable and elegant.

Exercises:

- Confirm your source code for the last section[112]
  - Confirm the changes from the last section[113]
- Refactor the Table and Button component to stateless functional components
- Read about ES6 class components and functional stateless components[114]

[112] http://bit.ly/2HpBVzR
[113] http://bit.ly/2Hs4tsR
[114] https://reactjs.org/docs/components-and-props.html

Styling Components

In this section, we'll add some basic styling to our application and components using the *src/App.css* and *src/index.css* files. These files should already be in your project, since you have bootstrapped it with *create-react-app*. They should be imported in your *src/App.js* and *src/index.js* files too. The following is CSS that can be copied and pasted to these files, but feel free to use your own if you're comfortable with CSS.

First, styling for your overall application:

src/index.css

```
body {
  color: #222;
  background: #f4f4f4;
  font: 400 14px CoreSans, Arial, sans-serif;
}

a {
  color: #222;
}

a:hover {
  text-decoration: underline;
}

ul, li {
  list-style: none;
  padding: 0;
  margin: 0;
}

input {
  padding: 10px;
  border-radius: 5px;
  margin-right: 10px;
  border: 1px solid #dddddd;
}

button {
  padding: 10px;
  border-radius: 5px;
  border: 1px solid #dddddd;
  background: transparent;
```

```
  color: #808080;
  cursor: pointer;
}

button:hover {
  color: #222;
}
```

Second, styling for your components in the App file:

src/App.css

```
.page {
  margin: 20px;
}

.interactions {
  text-align: center;
}

.table {
  margin: 20px 0;
}

.table-header {
  display: flex;
  line-height: 24px;
  font-size: 16px;
  padding: 0 10px;
  justify-content: space-between;
}

.table-empty {
  margin: 200px;
  text-align: center;
  font-size: 16px;
}

.table-row {
  display: flex;
  line-height: 24px;
  white-space: nowrap;
  margin: 10px 0;
  padding: 10px;
```

```
  background: #ffffff;
  border: 1px solid #e3e3e3;
}

.table-header > span {
  overflow: hidden;
  text-overflow: ellipsis;
  padding: 0 5px;
}

.table-row > span {
  overflow: hidden;
  text-overflow: ellipsis;
  padding: 0 5px;
}

.button-inline {
  border-width: 0;
  background: transparent;
  color: inherit;
  text-align: inherit;
  -webkit-font-smoothing: inherit;
  padding: 0;
  font-size: inherit;
  cursor: pointer;
}

.button-active {
  border-radius: 0;
  border-bottom: 1px solid #38BB6C;
}
```

Now we can use this style with some of our components. Remember to use React `className` instead of `class` as an HTML attribute.

First, apply it in your App ES6 class component:

src/App.js

```
class App extends Component {

  ...

  render() {
    const { searchTerm, list } = this.state;
    return (
      <div className="page">
        <div className="interactions">
          <Search
            value={searchTerm}
            onChange={this.onSearchChange}
          >
            Search
          </Search>
        </div>
        <Table
          list={list}
          pattern={searchTerm}
          onDismiss={this.onDismiss}
        />
      </div>
    );
  }
}
```

Second, apply it in your Table functional stateless component:

src/App.js

```
const Table = ({ list, pattern, onDismiss }) =>
  <div className="table">
    {list.filter(isSearched(pattern)).map(item =>
      <div key={item.objectID} className="table-row">
        <span>
          <a href={item.url}>{item.title}</a>
        </span>
        <span>{item.author}</span>
        <span>{item.num_comments}</span>
        <span>{item.points}</span>
        <span>
          <Button
```

```
            onClick={() => onDismiss(item.objectID)}
            className="button-inline"
          >
            Dismiss
          </Button>
        </span>
      </div>
    )}
  </div>
```

Now the application and components have been styled with basic CSS. Further, we know JSX mixes up HTML and JavaScript, and now we could arguably add CSS to that mix. That's called inline style, where you can define JavaScript objects and pass them to the style attribute of an element.

Let's keep the Table column width flexible by using inline style.

src/App.js

```
const Table = ({ list, pattern, onDismiss }) =>
  <div className="table">
    {list.filter(isSearched(pattern)).map(item =>
      <div key={item.objectID} className="table-row">
        <span style={{ width: '40%' }}>
          <a href={item.url}>{item.title}</a>
        </span>
        <span style={{ width: '30%' }}>
          {item.author}
        </span>
        <span style={{ width: '10%' }}>
          {item.num_comments}
        </span>
        <span style={{ width: '10%' }}>
          {item.points}
        </span>
        <span style={{ width: '10%' }}>
          <Button
            onClick={() => onDismiss(item.objectID)}
            className="button-inline"
          >
            Dismiss
          </Button>
        </span>
      </div>
    )}
  </div>
```

The style is inlined now. Define the style objects outside of your elements to make it cleaner.

Code Playground

```
const largeColumn = {
  width: '40%',
};

const midColumn = {
  width: '30%',
};

const smallColumn = {
  width: '10%',
};
```

After that, we use them in the columns: `<span style={smallColumn}>`. There different opinions and solutions about style in React, but the pure inline CSS we used is sufficient for this tutorial. I don't want to be opinionated here, but I want to leave you some more options. You can read about them and apply them on your own:

- styled-components[115]
- CSS Modules[116] (read my short article on how to use CSS modules in create-react-app[117])
- Sass[118] (read my short article on how to use Sass in create-react-app[119])
- UI Libraries: Semantic UI[120] or Material UI

But if you are new to React, I would recommend to stick to pure CSS and inline style for now.

Exercises:

- Confirm your source code for the last section[121]
  - Confirm the changes from the last section[122]

[115] https://github.com/styled-components/styled-components
[116] https://github.com/css-modules/css-modules
[117] https://www.robinwieruch.de/create-react-app-css-modules/
[118] https://sass-lang.com/
[119] https://www.robinwieruch.de/create-react-app-with-sass-support/
[120] https://www.robinwieruch.de/react-semantic-ui-tutorial/
[121] http://bit.ly/2HlA6Ed
[122] http://bit.ly/2HlA67b

You have learned the basics on how to write your own React application! Let's recap the last chapter:

- **React**
  - Use `this.state` and `setState()` to manage your local component state
  - Pass functions or class methods to your element handler
  - Use forms and events in React to add interactions
  - Unidirectional data flow is an important concept in React
  - Embrace controlled components
  - Compose components with children and reusable components
  - Usage and implementation of ES6 class components and functional stateless components
  - Approaches to style your components
- **ES6**
  - Functions that are bound to a class are class methods
  - Destructuring of objects and arrays
  - Default parameters
- **General**
  - Higher-order functions

Again, it makes sense to take a break, internalize the lessons, and apply them on your own. Experiment with the source code you have written so far. The source code for this project is found in the official repository[123].

[123] http://bit.ly/2HlA6Ed

Getting Real with APIs

Now it's time to get real with APIs and move past sample data. If you are not familiar, I encourage you to read my article on how I got to know APIs[124].

For our first foray into the concept, we will be using Hacker News[125], a solid news aggregator about tech topics. In this exercise, we will use the Hacker News API to fetch trending stories. There are basic[126] and search[127] APIs to get data from the platform. Search makes sense in this application, because we want to be able to search Hacker News stories. Visit the API specification to get an understanding of the data structure.

[124] https://www.robinwieruch.de/what-is-an-api-javascript/
[125] https://news.ycombinator.com/
[126] https://github.com/HackerNews/API
[127] https://hn.algolia.com/api

Lifecycle Methods

You may remember lifecycle methods were mentioned briefly in the last chapter, as a hook into the lifecycle of a React component. They can be used in ES6 class components, but not in functional stateless components. Besides the `render()` method, there are several methods that can be overridden in a React ES6 class component. All of these are the lifecycle methods.

We have already covered two lifecycle methods that can be used in an ES6 class component:

- The constructor is only called when an instance of the component is created and inserted in the DOM. The component gets instantiated in a process called mounting.
- The `render()` method is called during the mount process too, but also when the component updates. Each time the state or the props of a component changes, the `render()` method is called.

There are two more lifecycle methods when mounting a component: `getDerivedStateFromProps()` and `componentDidMount()`. The constructor is called first, `getDerivedStateFromProps()` is called before the `render()` method, and `componentDidMount()` is called after the `render()` method.

Overall, the mounting process has `4` lifecycle methods, invoked in the following order:

- constructor()
- getDerivedStateFromProps()
- render()
- componentDidMount()

For the update lifecycle of a component when the state or the props change, there are `5` lifecycle methods, in the following order:

- getDerivedStateFromProps()
- shouldComponentUpdate()
- render()
- getSnapshotBeforeUpdate()
- componentDidUpdate()

Lastly, there is the unmounting lifecycle. It has only one lifecycle method: `componentWillUnmount()`.

You don't need to know all the lifecycle methods from the beginning, and even in a large React application you'll only use a few of them besides the `constructor()` and the `render()` methods. Still, it is good to know each lifecycle method can be used for specific purposes:

- **constructor(props)** is called when the component gets initialized. You can set an initial component state and bind class methods during that lifecycle method.

- **static getDerivedStateFromProps(props, state)** is called before the `render()` lifecycle method, both on the initial mount and on the subsequent updates. It should return an object to update the state, or null to update nothing. It exists for **rare** use cases where the state depends on changes in props over time. It is important to know that this is a static method and it doesn't have access to the component instance.
- **render()** is a mandatory lifecycle method that returns elements as an output of the component. The method should be pure, so it shouldn't modify the component state. It gets an input as props and state, and returns an element.
- **componentDidMount()** is called once, when the component mounted. That's the perfect time to do an asynchronous request to fetch data from an API. The fetched data is stored in the local component state to display it in the `render()` lifecycle method.
- **shouldComponentUpdate(nextProps, nextState)** is always called when the component updates due to state or props changes. You will use it in mature React applications for performance optimization. Depending on a boolean that you return from this lifecycle method, the component and all its children will render or will not render on an update lifecycle. You can prevent the render lifecycle method of a component.
- **getSnapshotBeforeUpdate(prevProps, prevState)** is a lifecycle method, invoked before the most recently rendered output is committed to the DOM. In rare cases, the component needs to capture information from the DOM before it is potentially changed. This lifecycle method enables the component to do it. Another method (`componentDidUpdate()`) will receive any value returned by `getSnapshotBeforeUpdate()` as a parameter.
- **componentDidUpdate(prevProps, prevState, snapshot)** is a lifecycle method that is invoked immediately after updating, but not for the initial render. You can use it as to perform DOM operations or to perform more asynchronous requests. If your component implements the `getSnapshotBeforeUpdate()` method, the value it returns will be received as the `snapshot` parameter.
- **componentWillUnmount()** is called before you destroy your component. You can use this lifecycle method to perform any clean up tasks.

As you may have gathered, the `constructor()` and `render()` lifecycle methods are the most commonly used lifecycle methods for ES6 class components. The `render()` method is always required to return a component instance.

Lastly, `componentDidCatch(error, info)` was introduced in React 16[128] as a way to catch errors in components. For instance, displaying the sample list in your application works fine, but there could be a time when a list in the local state is set to `null` by accident (e.g. when fetching the list from an external API, but the request failed and you set the local state of the list to null). It becomes impossible to filter and map the list, because it is `null` and not an empty list. The component would be broken, and the whole application would fail. Using `componentDidCatch()`, you can catch the error, store it in your local state, and show a message to the user.

[128] https://www.robinwieruch.de/what-is-new-in-react-16/

Exercises:

- Read about lifecycle methods in React[129]
- Read about the state related to lifecycle methods in React[130]
- Read about error handling in components[131]

[129] https://reactjs.org/docs/react-component.html
[130] https://reactjs.org/docs/state-and-lifecycle.html
[131] https://reactjs.org/blog/2017/07/26/error-handling-in-react-16.html

Fetching Data

Now we're prepared to fetch data from the Hacker News API. There was one lifecycle method mentioned that can be used to fetch data: `componentDidMount()`. Before we use it, let's set up the URL constants and default parameters to break the URL endpoint for the API request into smaller pieces.

src/App.js

```
import React, { Component } from 'react';
import './App.css';

const DEFAULT_QUERY = 'redux';

const PATH_BASE = 'https://hn.algolia.com/api/v1';
const PATH_SEARCH = '/search';
const PARAM_SEARCH = 'query=';

...
```

In JavaScript ES6, you can use template literals[132] for string concatenation or interpolation. You will use it to concatenate your URL for the API endpoint.

Code Playground

```
// ES6
const url = `${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${DEFAULT_QUERY}`;

// ES5
var url = PATH_BASE + PATH_SEARCH + '?' + PARAM_SEARCH + DEFAULT_QUERY;

console.log(url);
// output: https://hn.algolia.com/api/v1/search?query=redux
```

This will keep your URL composition flexible in the future. Below, the entire data fetch process will be presented, and each step will be explained afterward.

[132] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Template_literals

src/App.js

```
...

class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      result: null,
      searchTerm: DEFAULT_QUERY,
    };

    this.setSearchTopStories = this.setSearchTopStories.bind(this);
    this.onSearchChange = this.onSearchChange.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
  }

  setSearchTopStories(result) {
    this.setState({ result });
  }

  componentDidMount() {
    const { searchTerm } = this.state;

    fetch(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}`)
      .then(response => response.json())
      .then(result => this.setSearchTopStories(result))
      .catch(error => error);
  }

  ...
}
```

First, we remove the sample list of items, because we will return a real list from the Hacker News API, so the sample data is no longer used. The initial state of your component has an empty result and default search term now. The same default search term is used in the input field of the Search component, and in your first request.

Second, you use the `componentDidMount()` lifecycle method to fetch the data after the component mounted. The first fetch uses default search term from the local state. It will fetch "redux" related stories, because that is the default parameter.

Third, the native fetch API is used. The JavaScript ES6 template strings allow it to compose the URL with the `searchTerm`. The URL is the argument for the native fetch API function. The response is transformed to a JSON data structure, a mandatory step in a native fetch with JSON data structures, after which it can be set as result in the local component state. If an error occurs during the request, the function will run into the catch block instead of the then block.

Last, remember to bind your new component method in the constructor.

Now you can use the fetched data instead of the sample list. Note that the result is not only a list of data, but a complex object with meta information and a list of hits that are news stories[133]. You can output the local state with `console.log(this.state);` in your `render()` method to visualize it.

In the next step, we use the result to render it. But we will prevent it from rendering anything, so we will return null, when there is no result in the first place. Once the request to the API has succeeded, the result is saved to the state and the App component will re-render with the updated state.

src/App.js

```
class App extends Component {

  ...

  render() {
    const { searchTerm, result } = this.state;

    if (!result) { return null; }

    return (
      <div className="page">
        ...
        <Table
          list={result.hits}
          pattern={searchTerm}
          onDismiss={this.onDismiss}
        />
      </div>
    );
  }
}
```

Now, let's recap what happens during the component lifecycle. Your component is initialized by the constructor, after which it renders for the first time. We prevented it from displaying anything, because the result in the local state is null. It is allowed to return null for a component to display nothing. Then the `componentDidMount()` lifecycle method fetches the data from the

[133] https://hn.algolia.com/api

Hacker News API asynchronously. Once the data arrives, it changes your local component state in `setSearchTopStories()`. The update lifecycle activates because the local state was updated. The component runs the `render()` method again, but this time with populated result in your local component state. The component and the Table component will be rendered with its content.

We used the native fetch API supported by most browsers to perform an asynchronous request to an API. The *create-react-app* configuration makes sure it is supported by all browsers. There are also third-party node packages that you can use to substitute the native fetch API: axios[134]. You will use axios later in this book.

In this book, we build on JavaScript's shorthand notation for truthfulness checks. In the previous example, `if (!result)` was used in favor of `if (result === null)`. The same applies for other cases as well. For instance, `if (!list.length)` is used in favor of `if (list.length === 0)` or `if (someString)` is used in favor of `if (someString !== '')`.

The list of hits should now be visible in our application; however, two regression bugs have appeared. First, the "Dismiss" button is broken, because it doesn't know about the complex result object, but it still operates on the plain list from the sample data when dismissing an item. Second, when the list is displayed and you try to search for something else, it gets filtered on the client-side, though the initial search was made by searching for stories on the server-side. The perfect behavior would be to fetch another result object from the API when using the Search component. Both regression bugs will be fixed in the following chapters.

Exercises:

- Confirm your source code for the last section[135]
  - Confirm the changes from the last section[136]
- Read about ES6 template literals[137]
- Read about the native fetch API[138]
- Read about data fetching in React[139]

[134] https://github.com/mzabriskie/axios
[135] http://bit.ly/2HkOIDO
[136] http://bit.ly/2HkOHzK
[137] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Template_literals
[138] https://developer.mozilla.org/en/docs/Web/API/Fetch_API
[139] https://www.robinwieruch.de/react-fetching-data/

ES6 Spread Operators

The “Dismiss” button doesn’t work because the `onDismiss()` method is not aware of the complex result object. It only knows about a plain list in the local state. But it isn’t a plain list anymore. Let’s change it to operate on the result object instead of the list itself.

src/App.js

```
onDismiss(id) {
  const isNotId = item => item.objectID !== id;
  const updatedHits = this.state.result.hits.filter(isNotId);
  this.setState({
    ...
  });
}
```

In `setState()`, the result is now a complex object, and the list of hits is only one of multiple properties in the object. Only the list gets updated when an item gets removed in the result object, though, while the other properties stay the same.

We could alleviate this challenge by mutating the hits in the result object. It is demonstrated below for the sake of understanding, but we’ll end up using another way.

Code Playground

```
// don't do this
this.state.result.hits = updatedHits;
```

As we know, React embraces immutable data structures, so we don’t want to mutate an object (or mutate the state directly). We want to generate a new object based on the information given, so none of the objects get altered and we keep the immutable data structures. You will always return a new object, but never alter the original object.

For this, we use JavaScript ES6’s `Object.assign()`. It takes a target object as first argument. All following arguments are source objects, which are merged into the target object. The target object can be an empty object. It embraces immutability, because no source object gets mutated.

Code Playground

```
const updatedHits = { hits: updatedHits };
const updatedResult = Object.assign({}, this.state.result, updatedHits);
```

Source objects will override previously merged objects with the same property names. Let’s do this on the `onDismiss()` method:

src/App.js

```
onDismiss(id) {
  const isNotId = item => item.objectID !== id;
  const updatedHits = this.state.result.hits.filter(isNotId);
  this.setState({
    result: Object.assign({}, this.state.result, { hits: updatedHits })
  });
}
```

That's one solution, but there is a simpler way in JavaScript ES6, called the *spread operator*. The spread operator is shown with three dots: . . . When it is used, every value from an array or object gets copied to another array or object.

We'll examine the ES6 **array** spread operator even though you don't need it yet:

Code Playground

```
const userList = ['Robin', 'Andrew', 'Dan'];
const additionalUser = 'Jordan';
const allUsers = [ ...userList, additionalUser ];

console.log(allUsers);
// output: ['Robin', 'Andrew', 'Dan', 'Jordan']
```

The `allUsers` variable is a completely new array. The other variables `userList` and `additionalUser` stay the same. You can even merge two arrays into a new array.

Code Playground

```
const oldUsers = ['Robin', 'Andrew'];
const newUsers = ['Dan', 'Jordan'];
const allUsers = [ ...oldUsers, ...newUsers ];

console.log(allUsers);
// output: ['Robin', 'Andrew', 'Dan', 'Jordan']
```

Now let's have a look at the object spread operator, which is not JavaScript ES6. It is part of a proposal for a next JavaScript version[140] that is already being used by the React community, which is why *create-react-app* incorporated the feature in the configuration. The object spread operator works just like the JavaScript ES6 array spread operator, except with objects. Each key value pair is copied into a new object:

[140] https://github.com/sebmarkbage/ecmascript-rest-spread

Code Playground

```
const userNames = { firstname: 'Robin', lastname: 'Wieruch' };
const age = 28;
const user = { ...userNames, age };

console.log(user);
// output: { firstname: 'Robin', lastname: 'Wieruch', age: 28 }
```

Multiple objects can be spread, as with the array spread example.

Code Playground

```
const userNames = { firstname: 'Robin', lastname: 'Wieruch' };
const userAge = { age: 28 };
const user = { ...userNames, ...userAge };

console.log(user);
// output: { firstname: 'Robin', lastname: 'Wieruch', age: 28 }
```

It can be used to replace `Object.assign()`:

src/App.js

```
onDismiss(id) {
  const isNotId = item => item.objectID !== id;
  const updatedHits = this.state.result.hits.filter(isNotId);
  this.setState({
    result: { ...this.state.result, hits: updatedHits }
  });
}
```

The "Dismiss" button should work now, because the `onDismiss()` method is aware of the complex result object, and how to update it after dismissing an item from the list.

Exercises:

- Confirm your source code for the last section[141]
  - Confirm the changes from the last section[142]
- Read about the ES6 Object.assign()[143]
- Read about the ES6 array (and object) spread operator[144]

[141] http://bit.ly/2HmzEpb
[142] http://bit.ly/2HpC32j
[143] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Object/assign
[144] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Spread_operator

Conditional Rendering

Conditional rendering is usually introduced early in React applications, though not in our working example. It happens when you decide to render either of two elements, which sometimes is a choice between rendering an element or nothing. The simplest usage of a conditional rendering can be expressed by an if-else statement in JSX.

The `result` object in the local component state is `null` in the beginning. So far, the App component returned no elements when the `result` hasn't arrived from the API. That's already a conditional rendering, because you return earlier from the `render()` lifecycle method for a certain condition. The App component either renders nothing or its elements.

But let's go one step further. It makes more sense to wrap the Table component, which is the only component that depends on `result` in an independent conditional rendering. Everything else should be displayed, even though there is no `result` yet. You can simply use a ternary operator[145] in the JSX:

src/App.js

```
class App extends Component {

  ...

  render() {
    const { searchTerm, result } = this.state;
    return (
      <div className="page">
        <div className="interactions">
          <Search
            value={searchTerm}
            onChange={this.onSearchChange}
          >
            Search
          </Search>
        </div>
        { result
          ? <Table
            list={result.hits}
            pattern={searchTerm}
            onDismiss={this.onDismiss}
          />
          : null
        }
```

[145] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_Operator

```
      </div>
    );
  }
}
```

That's your second option to express a conditional rendering. A third option is the logical `&&` operator. In JavaScript a `true && 'Hello World'` always evaluates to 'Hello World'. A `false && 'Hello World'` always evaluates to false.

Code Playground

```
const result = true && 'Hello World';
console.log(result);
// output: Hello World

const result = false && 'Hello World';
console.log(result);
// output: false
```

In React, you can make use of that behavior. If the condition is true, the expression after the logical `&&` operator will be the output. If the condition is false, React ignores the expression. It is applicable in the Table component's conditional rendering case, because it should either return a Table or nothing.

src/App.js

```
{ result &&
  <Table
    list={result.hits}
    pattern={searchTerm}
    onDismiss={this.onDismiss}
  />
}
```

These were a few approaches to use conditional rendering in React. You can read about more alternatives in an exhaustive list of examples[146]. Moreover, you will get to know their different uses and when to apply them.

You should be able to see the fetched data in your application by now. Everything except the Table is displayed when data fetching is pending. Once the request resolves the result and stores it into the local state, the Table is displayed because the `render()` method runs again, and the condition in the conditional rendering resolves in favor of displaying the Table component.

[146] https://www.robinwieruch.de/conditional-rendering-react/

Exercises:

- Confirm your source code for the last section[147]
  - Confirm the changes from the last section[148]
- Read about different ways for conditional renderings[149]
- Read about React conditional rendering[150]

[147] http://bit.ly/2HnZObc
[148] http://bit.ly/2Ho99jl
[149] https://www.robinwieruch.de/conditional-rendering-react/
[150] https://reactjs.org/docs/conditional-rendering.html

Client- or Server-side Search

Now, when you use the Search component with its input field, you will filter the list. That's happening on the client-side, though. We want to use the Hacker News API to search on the server-side. Otherwise you would only deal with the first API response you got on `componentDidMount()`, the default search term parameter.

You can define an `onSearchSubmit()` method in your App component, which fetches results from the Hacker News API when executing a search in the Search component.

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      result: null,
      searchTerm: DEFAULT_QUERY,
    };

    this.setSearchTopStories = this.setSearchTopStories.bind(this);
    this.onSearchChange = this.onSearchChange.bind(this);
    this.onSearchSubmit = this.onSearchSubmit.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
  }

  ...

  onSearchSubmit() {
    const { searchTerm } = this.state;
  }

  ...
}
```

The `onSearchSubmit()` method should use the same functionality as the `componentDidMount()` lifecycle method, but this time with a modified search term from the local state and not with the initial default search term. Thus you can extract the functionality as a reusable class method.

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      result: null,
      searchTerm: DEFAULT_QUERY,
    };

    this.setSearchTopStories = this.setSearchTopStories.bind(this);
    this.fetchSearchTopStories = this.fetchSearchTopStories.bind(this);
    this.onSearchChange = this.onSearchChange.bind(this);
    this.onSearchSubmit = this.onSearchSubmit.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
  }

  ...

  fetchSearchTopStories(searchTerm) {
    fetch(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}`)
      .then(response => response.json())
      .then(result => this.setSearchTopStories(result))
      .catch(error => error);
  }

  componentDidMount() {
    const { searchTerm } = this.state;
    this.fetchSearchTopStories(searchTerm);
  }

  ...

  onSearchSubmit() {
    const { searchTerm } = this.state;
    this.fetchSearchTopStories(searchTerm);
  }

  ...
}
```

Now the Search component has to add an additional button to trigger the search request explicitly. Otherwise, we'd fetch data from the Hacker News API every time the input field changes. We want to do it explicitly in a `onClick()` handler.

As an alternative, you could debounce (delay) the `onChange()` function and spare the button, but it would add more complexity, and we want to keep things simple for now.

First, pass the `onSearchSubmit()` method to your Search component.

src/App.js

```
class App extends Component {

  ...

  render() {
    const { searchTerm, result } = this.state;
    return (
      <div className="page">
        <div className="interactions">
          <Search
            value={searchTerm}
            onChange={this.onSearchChange}
            onSubmit={this.onSearchSubmit}
          >
            Search
          </Search>
        </div>
        { result &&
          <Table
            list={result.hits}
            pattern={searchTerm}
            onDismiss={this.onDismiss}
          />
        }
      </div>
    );
  }
}
```

Second, introduce a button in your Search component. The button has the `type="submit"` and the form uses its `onSubmit` attribute to pass the `onSubmit()` method. You can reuse the `children` property, but this time it will be used as the content of the button.

src/App.js

```
const Search = ({
  value,
  onChange,
  onSubmit,
  children
}) =>
  <form onSubmit={onSubmit}>
    <input
      type="text"
      value={value}
      onChange={onChange}
    />
    <button type="submit">
      {children}
    </button>
  </form>
```

In the Table, you can remove the filter functionality, because there will be no client-side filter (search) anymore. Don't forget to remove the `isSearched()` function as well, as it won't be used anymore. Now, the result comes directly from the Hacker News API when the user clicks the "Search" button.

src/App.js

```
class App extends Component {

  ...

  render() {
    const { searchTerm, result } = this.state;
    return (
      <div className="page">
        ...
        { result &&
          <Table
            list={result.hits}
            onDismiss={this.onDismiss}
          />
        }
      </div>
    );
  }
}
```

```
...

const Table = ({ list, onDismiss }) =>
  <div className="table">
    {list.map(item =>
      ...
    )}
  </div>
```

Now when you try to search, you will notice the browser reloads. That's a native browser behavior for a submit callback in an HTML form. In React, you will often come across the `preventDefault()` event method to suppress the native browser behavior.

src/App.js

```
onSearchSubmit(event) {
  const { searchTerm } = this.state;
  this.fetchSearchTopStories(searchTerm);
  event.preventDefault();
}
```

You should be able to search different Hacker News stories now. We have interacted with a real API, and there should be no more client-side searches.

Exercises:

- Confirm your source code for the last section[151]
  - Confirm the changes from the last section[152]
- Read about synthetic events in React[153]
- Experiment with the Hacker News API[154]

[151] http://bit.ly/2HtekOK
[152] http://bit.ly/2HnYg0J
[153] https://reactjs.org/docs/events.html
[154] https://hn.algolia.com/api

Paginated Fetch

Take a closer look at the data structure and observe how the Hacker News API[155] returns more than a list of hits. Precisely it returns a paginated list. The page property, which is `0` in the first response, can be used to fetch more paginated sublists as results. You only need to pass the next page with the same search term to the API.

Let's extend the composable API constants so it can deal with paginated data:

src/App.js

```
const DEFAULT_QUERY = 'redux';

const PATH_BASE = 'https://hn.algolia.com/api/v1';
const PATH_SEARCH = '/search';
const PARAM_SEARCH = 'query=';
const PARAM_PAGE = 'page=';
```

Now you can use the new constant to add the page parameter to your API request:

Code Playground

```
const url = `${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${DEFAULT_QUERY}&${PARAM_PAGE\
}`;

console.log(url);
// output: https://hn.algolia.com/api/v1/search?query=redux&page=
```

The `fetchSearchTopStories()` method will take the page as second argument. If you don't provide the second argument, it will fallback to the `0` page for the initial request. Thus the `componentDidMount()` and `onSearchSubmit()` methods fetch the first page on the first request. Every additional fetch should fetch the next page by providing the second argument.

src/App.js

```
class App extends Component {

  ...

  fetchSearchTopStories(searchTerm, page = 0) {
    fetch(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}${pa\
ge}`)
      .then(response => response.json())
```

[155] https://hn.algolia.com/api

```
      .then(result => this.setSearchTopStories(result))
      .catch(error => error);
  }

  ...

}
```

The page argument uses the JavaScript ES6 default parameter to introduce the fallback to page `0` in case no defined page argument is provided for the function.

Now you can use the current page from the API response in `fetchSearchTopStories()`. You can use this method in a button to fetch more stories with an `onClick` button handler. Let's use the Button to fetch more paginated data from the Hacker News API. For this, we'll define the `onClick()` handler, which takes the current search term and the next page (current page + 1).

src/App.js

```
class App extends Component {

  ...

  render() {
    const { searchTerm, result } = this.state;
    const page = (result && result.page) || 0;
    return (
      <div className="page">
        <div className="interactions">
        ...
        { result &&
          <Table
            list={result.hits}
            onDismiss={this.onDismiss}
          />
        }
        <div className="interactions">
          <Button onClick={() => this.fetchSearchTopStories(searchTerm, page + 1)}>
            More
          </Button>
        </div>
      </div>
    );
  }
}
```

In your `render()` method, make sure to default to page `0` when there is no result. Remember, the `render()` method is called before the data is fetched asynchronously in the `componentDidMount()` lifecycle method.

There is still one step missing, because fetching the next page of data will override your previous page of data. We want to concatenate the old and new list of hits from the local state and new result object, so we'll adjust its functionality to add new data rather than override it.

src/App.js

```
setSearchTopStories(result) {
  const { hits, page } = result;

  const oldHits = page !== 0
    ? this.state.result.hits
    : [];

  const updatedHits = [
    ...oldHits,
    ...hits
  ];

  this.setState({
    result: { hits: updatedHits, page }
  });
}
```

A couple things happen in the `setSearchTopStories()` method now. First, you get the hits and page from the result.

Second, you have to check if there are already old hits. When the page is `0`, it is a new search request from `componentDidMount()` or `onSearchSubmit()`. The hits are empty. But when you click the "More" button to fetch paginated data the page isn't `0`. The old hits are already stored in your state and thus can be used.

Third, you don't want to override the old hits. You can merge old and new hits from the recent API request, which can be done with a JavaScript ES6 array spread operator.

Fourth, you set the merged hits and page in the local component state.

Now we'll make one last adjustment. When you try the "More" button it only fetches a few list items. The API URL can be extended to fetch more list items with each request, so we add even more composable path constants:

src/App.js

```
const DEFAULT_QUERY = 'redux';
const DEFAULT_HPP = '100';

const PATH_BASE = 'https://hn.algolia.com/api/v1';
const PATH_SEARCH = '/search';
const PARAM_SEARCH = 'query=';
const PARAM_PAGE = 'page=';
const PARAM_HPP = 'hitsPerPage=';
```

Now you can use the constants to extend the API URL.

src/App.js

```
fetchSearchTopStories(searchTerm, page = 0) {
  // be careful with the "\" which shows up in the PDF/print version of the book
  // it's only a line break a should not be in the actual code
  // https://github.com/the-road-to-learn-react/the-road-to-learn-react/issues/43
  fetch(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}${page\
}&${PARAM_HPP}${DEFAULT_HPP}`)
    .then(response => response.json())
    .then(result => this.setSearchTopStories(result))
    .catch(error => error);
}
```

The request to the Hacker News API fetches more list items in one request than before. As you can see, a powerful API such as the Hacker News API gives plenty of ways to experiment with real world data. You should make use of it to make your endeavours when learning something new more exciting. That's how I learned about the empowerment that APIs provide[156] when learning a new programming language or library.

Exercises:

- Confirm your source code for the last section[157]
  - Confirm the changes from the last section[158]
- Read about ES6 default parameters[159]
- Experiment with the Hacker News API parameters[160]

[156] https://www.robinwieruch.de/what-is-an-api-javascript/
[157] http://bit.ly/2Hotq8c
[158] http://bit.ly/2Hmrj52
[159] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/Default_parameters
[160] https://hn.algolia.com/api

Client Cache

Each search submit makes a request to the Hacker News API. You might search for "redux", followed by "react" and eventually "redux" again. In total it makes 3 requests. But you searched for "redux" twice and both times it took a whole asynchronous roundtrip to fetch the data. In a client-sided cache, you would store each result. When a request to the API is made, it checks if a result is already there and uses the cache if it is. Otherwise an API request is made to fetch the data.

To have a client cache for each result, you have to store multiple `results` rather than one `result` in your local component state. The results object will be a map with the search term as key and the result as value. Each result from the API will be saved by the search term (key).

At the moment, your result in the local state looks like this:

Code Playground

```
result: {
  hits: [ ... ],
  page: 2,
}
```

Imagine you have made two API requests. One for the search term "redux" and another one for "react". The results object should look like the following:

Code Playground

```
results: {
  redux: {
    hits: [ ... ],
    page: 2,
  },
  react: {
    hits: [ ... ],
    page: 1,
  },
  ...
}
```

Let's implement a client-side cache with React `setState()`. First, rename the `result` object to `results` in the initial component state. Second, define a temporary `searchKey` to store each `result`.

src/App.js

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      results: null,
      searchKey: '',
      searchTerm: DEFAULT_QUERY,
    };

    ...

  }

  ...

}
```

The `searchKey` has to be set before each request is made. It reflects the `searchTerm`. Understanding why we don't use `searchTerm` here is a crucial part to understand before continuing with the implementation. The `searchTerm` is a fluctuant variable, because it gets changed every time you type into the search input field. We want a non fluctuant variable that determines the recent submitted search term to the API and can be used to retrieve the correct result from the map of results. It is a pointer to your current result in the cache, which can be used to display the current result in the `render()` method.

src/App.js

```
componentDidMount() {
  const { searchTerm } = this.state;
  this.setState({ searchKey: searchTerm });
  this.fetchSearchTopStories(searchTerm);
}

onSearchSubmit(event) {
  const { searchTerm } = this.state;
  this.setState({ searchKey: searchTerm });
  this.fetchSearchTopStories(searchTerm);
  event.preventDefault();
}
```

Now we will change where the result is stored to the local component state. It should store each result by `searchKey`.

src/App.js

```
class App extends Component {

  ...

  setSearchTopStories(result) {
    const { hits, page } = result;
    const { searchKey, results } = this.state;

    const oldHits = results && results[searchKey]
      ? results[searchKey].hits
      : [];

    const updatedHits = [
      ...oldHits,
      ...hits
    ];

    this.setState({
      results: {
        ...results,
        [searchKey]: { hits: updatedHits, page }
      }
    });
  }

  ...

}
```

The `searchKey` will be used as the key to save the updated hits and page in a `results` map.

First, you have to retrieve the `searchKey` from the component state. Remember that the `searchKey` gets set on `componentDidMount()` and `onSearchSubmit()`.

Second, the old hits have to get merged with the new hits as before. But this time the old hits get retrieved from the `results` map with the `searchKey` as key.

Third, a new result can be set in the `results` map in the state. Let's examine the `results` object in `setState()`.

src/App.js

```
results: {
  ...results,
  [searchKey]: { hits: updatedHits, page }
}
```

The bottom part makes sure to store the updated result by `searchKey` in the results map. The value is an object with a hits and page property. The `searchKey` is the search term. You already learned the `[searchKey]: ...` syntax. It is an ES6 computed property name. It helps you allocate values dynamically in an object.

The upper part needs to spread all other results by `searchKey` in the state using the object spread operator. Otherwise, you would lose all results that you have stored before.

Now you store all results by search term. That's the first step to enable your cache. In the next step, you can retrieve the result depending on the non fluctuant `searchKey` from your map of results. That's why you had to introduce the `searchKey` in the first place as non fluctuant variable. Otherwise, the retrieval would be broken when you would use the fluctuant `searchTerm` to retrieve the current result, because this value might change when we use the Search component.

src/App.js

```
class App extends Component {

  ...

  render() {
    const {
      searchTerm,
      results,
      searchKey
    } = this.state;

    const page = (
      results &&
      results[searchKey] &&
      results[searchKey].page
    ) || 0;

    const list = (
      results &&
      results[searchKey] &&
      results[searchKey].hits
    ) || [];
```

```
    return (
      <div className="page">
        <div className="interactions">
          ...
        </div>
        <Table
          list={list}
          onDismiss={this.onDismiss}
        />
        <div className="interactions">
          <Button onClick={() => this.fetchSearchTopStories(searchKey, page + 1)}>
            More
          </Button>
        </div>
      </div>
    );
  }
}
```

Since you default to an empty list when there is no result by `searchKey`, you can spare the conditional rendering for the Table component. Additionally, we need to pass the `searchKey` rather than the `searchTerm` to the "More" button. Otherwise, your paginated fetch depends on the `searchTerm` value which is fluctuant. Moreover make sure to keep the fluctuant `searchTerm` property for the input field in the Search component.

The search functionality should store all results from the Hacker News API now.

Now, we can see the `onDismiss()` method needs improvement, as it still deals with the `result` object. We want it deal with multiple `results`:

src/App.js

```
  onDismiss(id) {
    const { searchKey, results } = this.state;
    const { hits, page } = results[searchKey];

    const isNotId = item => item.objectID !== id;
    const updatedHits = hits.filter(isNotId);

    this.setState({
      results: {
        ...results,
        [searchKey]: { hits: updatedHits, page }
      }
```

```
    });
  }
```

Check to make sure the "Dismiss" button is working again.

Note that nothing will stop the application from sending an API request on each search submit. Though there might be already a result, there is no check that prevents the request, so the cache functionality is not complete yet. It caches the results, but it doesn't make use of them. The last step is to prevent the API request when a result is available in the cache:

src/App.js

```
class App extends Component {

  constructor(props) {

    ...

    this.needsToSearchTopStories = this.needsToSearchTopStories.bind(this);
    this.setSearchTopStories = this.setSearchTopStories.bind(this);
    this.fetchSearchTopStories = this.fetchSearchTopStories.bind(this);
    this.onSearchChange = this.onSearchChange.bind(this);
    this.onSearchSubmit = this.onSearchSubmit.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
  }

  needsToSearchTopStories(searchTerm) {
    return !this.state.results[searchTerm];
  }

  ...

  onSearchSubmit(event) {
    const { searchTerm } = this.state;
    this.setState({ searchKey: searchTerm });

    if (this.needsToSearchTopStories(searchTerm)) {
      this.fetchSearchTopStories(searchTerm);
    }

    event.preventDefault();
  }

  ...
```

```
}
```

Now your client only makes a request to the API once, though you searched for a term twice. Even paginated data with several pages gets cached that way, because you always save the last page for each result in the `results` map. This is a powerful approach to introduce caching into an application. The Hacker News API provides you with everything you need to cache paginated data effectively.

Error Handling

Now we've taken care of interactions with the Hacker News API. We've introduced an elegant way to cache results from the API and make use of its paginated list functionality to fetch an endless list of sublists of stories from the API. But no application is complete without error handling.

In this chapter, we introduce an efficient solution to add error handling for your application in case of an erroneous API request. We have learned the necessary building block to introduce error handling in React: local state and conditional rendering. The error is just another state, which we store in the local state and display with conditional rendering in the component.

Now, we'll implement it in the App component, since that's where we fetch data from the Hacker News API. First, introduce the error in the local state. It is initialized as null, but will be set to the error object in case of an error.

src/App.js

```
class App extends Component {
  constructor(props) {
    super(props);

    this.state = {
      results: null,
      searchKey: '',
      searchTerm: DEFAULT_QUERY,
      error: null,
    };

    ...
  }

...

}
```

Second, use the catch block in your native fetch to store the error object in the local state by using `setState()`. Every time the API request isn't successful, the catch block is executed.

src/App.js

```
class App extends Component {

  ...

  fetchSearchTopStories(searchTerm, page = 0) {
    // be careful with the "\" which shows up in the PDF/print version of the book
    // it's only a line break a should not be in the actual code
    // https://github.com/the-road-to-learn-react/the-road-to-learn-react/issues/43
    fetch(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}${pa\
ge}&${PARAM_HPP}${DEFAULT_HPP}`)
      .then(response => response.json())
      .then(result => this.setSearchTopStories(result))
      .catch(error => this.setState({ error }));
  }

  ...

}
```

Third, retrieve the error object from your local state in the `render()` method, and display a message in case of an error using React's conditional rendering.

src/App.js

```
class App extends Component {

  ...

  render() {
    const {
      searchTerm,
      results,
      searchKey,
      error
    } = this.state;

    ...

    if (error) {
      return <p>Something went wrong.</p>;
    }
```

```
    return (
      <div className="page">
        ...
      </div>
    );
  }
}
```

If you want to test that your error handling is working, change the API URL to something non-existent to force an error.

src/App.js

```
const PATH_BASE = 'https://hn.mydomain.com/api/v1';
```

Now you should see an error message instead of your application. It's up to you where you want to place the conditional rendering for the error message. In this case, the app is overridden by the error message, but that wouldn't be the best user experience. The remaining application would still be visible so the user knows it's still running:

src/App.js

```
class App extends Component {

  ...

  render() {
    const {
      searchTerm,
      results,
      searchKey,
      error
    } = this.state;

    const page = (
      results &&
      results[searchKey] &&
      results[searchKey].page
    ) || 0;

    const list = (
      results &&
      results[searchKey] &&
      results[searchKey].hits
```

```
    ) || [];

    return (
      <div className="page">
        <div className="interactions">
          ...
        </div>
        { error
          ? <div className="interactions">
            <p>Something went wrong.</p>
          </div>
          : <Table
            list={list}
            onDismiss={this.onDismiss}
          />
        }
        ...
      </div>
    );
  }
}
```

Remember to revert the URL for the API to the existent one:

src/App.js

```
const PATH_BASE = 'https://hn.algolia.com/api/v1';
```

The application now has error handling in case the API request fails.

Exercises:

- Confirm your source code for the last section[161]
  - Confirm the changes from the last section[162]
- Read about React's Error Handling for Components[163]

[161] http://bit.ly/2HoLAGY
[162] http://bit.ly/2HquJnd
[163] https://reactjs.org/blog/2017/07/26/error-handling-in-react-16.html

Axios instead of Fetch

Before, we introduced the native fetch API to perform a request to the Hacker News platform. The browser allows you to use this native fetch API. However, not all browsers support this, older browsers especially. Once you start testing your application in a headless browser environment (where there is no browser, it is mocked), there can be issues with the fetch API. There are a couple of ways to make fetch work in older browsers (polyfills) and in tests (isomorphic-fetch[164]), but these concepts are bit off-task for the purpose of learning React.

One alternative is to substitute the native fetch API with a stable library such as axios[165], which performs asynchronous requests to remote APIs. In this chapter, we will discover how to substitute a library, a native API of the browser in this case, with another library.

Below, the native fetch API is substituted with axios. First, we install axios on the command line:

Command Line

```
npm install axios
```

Second, import axios in your App component's file:

src/App.js

```
import React, { Component } from 'react';
import axios from 'axios';
import './App.css';

...
```

You can use axios instead of `fetch()`, and its usage looks almost identical to the native fetch API. It takes the URL as argument and returns a promise. You don't have to transform the returned response to JSON anymore, since axios wraps the result into a `data` object in JavaScript. Just make sure to adapt your code to the returned data structure:

[164] https://github.com/matthew-andrews/isomorphic-fetch
[165] https://github.com/axios/axios

src/App.js

```
class App extends Component {

  ...

  fetchSearchTopStories(searchTerm, page = 0) {
    axios(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}${pa\
ge}&${PARAM_HPP}${DEFAULT_HPP}`)
      .then(result => this.setSearchTopStories(result.data))
      .catch(error => this.setState({ error }));
  }

  ...

}
```

In this code, we call `axios()`, which uses an HTTP GET request by default. You can make the GET request explicit by calling `axios.get()`, or you can use another HTTP method such as HTTP POST with `axios.post()`. With these examples alone, we can see that axios is a powerful library to perform requests to remote APIs. I recommend you use it instead of the native fetch API when requests become complex, or you have to deal with promises.

There exists another improvement for the Hacker News request in the App component. Imagine the component mounts when the page is rendered for the first time in the browser. In `componentDidMount()` the component starts to make the request, but then the user navigates away from the page with this rendered component. Then the App component unmounts, but there is still a pending request from `componentDidMount()` lifecycle method. It will attempt to use `this.setState()` eventually in the `then()` or `catch()` block of the promise. You will likely see the following warning(s) on your command line, or in your browser's developer output:

- *Warning: Can only update a mounted or mounting component. This usually means you called setState, replaceState, or forceUpdate on an unmounted component. This is a no-op.*
- *Warning: Canâ€™t call setState (or forceUpdate) on an unmounted component. This is a no-op, but it indicates a memory leak in your application. To fix, cancel all subscriptions and asynchronous tasks in the componentWillUnmount method.*

If you encounter one of them in your browser's developer tools, checkout this walkthrough on how to prevent memory leaks in React[166]. It isn't too important for starting out with React, but I have seen many React beginners being confused about this warning.

This chapter has shown you how you can replace one library with another in React. If you run into any issues, you can use the vast library ecosystem in JavaScript to find a solution.

[166] https://www.robinwieruch.de/react-warning-cant-call-setstate-on-an-unmounted-component/

Exercises:

- Confirm your source code for the last section[167]
  - Confirm the changes from the last section[168]
- Read about why frameworks matter[169]

[167] http://bit.ly/2Hotuos
[168] http://bit.ly/2HkpMMC
[169] https://www.robinwieruch.de/why-frameworks-matter/

Now you've learned to interact with an API in React! Let's recap the last chapter:

- **React**
  - ES6 class component lifecycle methods for different use cases
  - `componentDidMount()` for API interactions
  - Conditional renderings
  - Synthetic events on forms
  - Error handling
  - Preventing `this.setState` in unmounted components
- **ES6 and beyond**
  - Template strings to compose strings
  - Spread operator for immutable data structures
  - Computed property names
  - Class fields
- **General**
  - Hacker News API interaction
  - Native fetch browser API
  - Client and server-side search
  - Pagination of data
  - Client-side caching
  - Axios as an alternative for the native fetch API

Again, it makes sense to take a break, internalize the lessons and apply them on your own. Experiment with the parameters for the API endpoint to query different results. You can find the source code in the official repository[170].

[170] http://bit.ly/2Hotuos

Code Organization and Testing

The chapter will focus on keeping code organized in a scaling application, so we will cover the best practices for structuring your folders and files. We will also cover testing, an important practice to keep source code robust, as well as useful tools for debugging a React application. This chapter will step back from the practical application for a moment, so we can cover these topics in detail before moving on.

ES6 Modules: Import and Export

In JavaScript ES6, you can import and export functionalities from modules. These can be functions, classes, components, constants, essentially anything you can assign to a variable. Modules can be single files or whole folders with one index file as entry point.

After we bootstrapped our application with *create-react-app* at the beginning, we already used several `import` and `export` statements in the initial files. The `import` and `export` statements help you to share code across multiple files. Historically there were already several solutions for this in the JavaScript environment, but it was a mess because there wasn't standardized method of performing this task. JavaScript ES6 added it as a native behavior eventually.

These statements embrace code splitting, where we distribute code across multiple files to keep it reusable and maintainable. The former is true because we can import a piece of code into multiple files. The latter is true because there is only one source where you maintain the piece of code.

We also want to think about code encapsulation, since not every functionality needs to be exported from a file. Some of these functionalities should only be used in files where they have been defined. File exports are basically a public API to a file, where only the exported functionalities are available to be reused elsewhere. This follows the best practice of encapsulation.

The following examples showcase the statements by sharing one or multiple variables across two files. In the end, the approach can scale to multiple files and could share more than simple variables.

The act of exporting one or multiple variables is called a named export:

Code Playground: file1.js

```
const firstname = 'Robin';
const lastname = 'Wieruch';

export { firstname, lastname };
```

And import them in another file with a relative path to the first file.

Code Playground: file2.js

```
import { firstname, lastname } from './file1.js';

console.log(firstname);
// output: Robin
```

You can also import all exported variables from another file as one object.

Code Playground: file2.js

```
import * as person from './file1.js';

console.log(person.firstname);
// output: Robin
```

Imports can have aliases, which are necessary when we import functionalities from multiple files that have the same named export.

Code Playground: file2.js

```
import { firstname as username } from './file1.js';

console.log(username);
// output: Robin
```

There is also the `default` statement, which can be used for a few cases:

- to export and import a single functionality
- to highlight the main functionality of the exported API of a module
- to have a fallback import functionality

Code Playground: file1.js

```
const robin = {
  firstname: 'Robin',
  lastname: 'Wieruch',
};

export default robin;
```

You have to leave out the curly braces to import the default export.

Code Playground: file2.js

```
import developer from './file1.js';

console.log(developer);
// output: { firstname: 'Robin', lastname: 'Wieruch' }
```

The import name can differ from the exported default name, and it can be used with the named export and import statements:

Code Playground: file1.js

```
const firstname = 'Robin';
const lastname = 'Wieruch';

const person = {
  firstname,
  lastname,
};

export {
  firstname,
  lastname,
};

export default person;
```

Import the default or the named exports in another file:

Code Playground: file2.js

```
import developer, { firstname, lastname } from './file1.js';

console.log(developer);
// output: { firstname: 'Robin', lastname: 'Wieruch' }
console.log(firstname, lastname);
// output: Robin Wieruch
```

You can spare the extra lines, and export the variables directly for named exports.

Code Playground: file1.js

```
export const firstname = 'Robin';
export const lastname = 'Wieruch';
```

These are the main functionalities for ES6 modules. They help you to organize your code, to maintain it, and to design reusable module APIs. You can also export and import functionalities to test them which you will do in a later chapter.

Exercises:

- Read about ES6 import[171]
- Read about ES6 export[172]

[171] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import
[172] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export

Code Organization with ES6 Modules

You might wonder why we didn't follow the best practices of putting components into different files for the *src/App.js* file. We already have multiple components in the file that can be defined in their own files/folders (modules). For the sake of learning React, it is practical to keep these items in one place. Once your React application grows, you should consider splitting these components into multiple modules so it can scale properly.

In the following, I propose several module structures you can apply. You can apply them as an exercise later, but we will continue developing our app with the *src/App.js* file to keep things simple.

Here is one possible module structure:

Folder Structure

```
src/
  index.js
  index.css
  App.js
  App.test.js
  App.css
  Button.js
  Button.test.js
  Button.css
  Table.js
  Table.test.js
  Table.css
  Search.js
  Search.test.js
  Search.css
```

This one separates the components into their own files, but it doesn't look too promising. You can see a lot of naming duplications, and only the file extension differs. Another module structure could be:

Folder Structure

```
src/
  index.js
  index.css
  App/
    index.js
    test.js
    index.css
  Button/
```

```
    index.js
    test.js
    index.css
  Table/
    index.js
    test.js
    index.css
  Search/
    index.js
    test.js
    index.css
```

Now it looks cleaner than before. The index naming of a file describes it as an entry point file to the folder. It is just a common naming convention, but you can use your own naming as well. In this module structure, a component is defined by its component declaration in the JavaScript file, but also by its style and tests. If you have trouble finding your components this way while searching for them in your editor/IDE, search for paths rather than files (e.g. search for "Table index").

Another step is extracting the constant variables from the App component, which were used to compose the Hacker News API URL.

Folder Structure

```
src/
  index.js
  index.css
  constants/
    index.js
  components/
    App/
      index.js
      test.js
      index.css
    Button/
      index.js
      test.js
      index.css
    ...
```

Naturally, the modules would split up into *src/constants/* and *src/components/*. The *src/constants/index.js* file could look like the following:

Code Playground: src/constants/index.js

```
export const DEFAULT_QUERY = 'redux';
export const DEFAULT_HPP = '100';
export const PATH_BASE = 'https://hn.algolia.com/api/v1';
export const PATH_SEARCH = '/search';
export const PARAM_SEARCH = 'query=';
export const PARAM_PAGE = 'page=';
export const PARAM_HPP = 'hitsPerPage=';
```

The *App/index.js* file could import these variables in order to use them.

Code Playground: src/components/App/index.js

```
import {
  DEFAULT_QUERY,
  DEFAULT_HPP,
  PATH_BASE,
  PATH_SEARCH,
  PARAM_SEARCH,
  PARAM_PAGE,
  PARAM_HPP,
} from '../../constants/index.js';

...
```

When you use the *index.js* naming convention, you can omit the filename from the relative path.

Code Playground: src/components/App/index.js

```
import {
  DEFAULT_QUERY,
  DEFAULT_HPP,
  PATH_BASE,
  PATH_SEARCH,
  PARAM_SEARCH,
  PARAM_PAGE,
  PARAM_HPP,
} from '../../constants';

...
```

The *index.js* file naming is convention was introduced in the Node.js world, where the index file is the entry point to a module. It describes the public API to the module. External modules are only allowed to use the *index.js* file to import shared code from the module. Consider the following module structure to demonstrate it:

Folder Structure

```
src/
  index.js
  App/
    index.js
  Buttons/
    index.js
    SubmitButton.js
    SaveButton.js
    CancelButton.js
```

The *Buttons/* folder has multiple button components defined in its distinct files. Each file can `export default` the specific component, making it available to *Buttons/index.js*. The *Buttons/index.js* file imports all different button representations and exports them as public module API.

Code Playground: src/Buttons/index.js

```
import SubmitButton from './SubmitButton';
import SaveButton from './SaveButton';
import CancelButton from './CancelButton';

export {
  SubmitButton,
  SaveButton,
  CancelButton,
};
```

Now the *src/App/index.js* can import the buttons from the public module API located in the *index.js* file.

Code Playground: src/App/index.js

```
import {
  SubmitButton,
  SaveButton,
  CancelButton
} from '../Buttons';
```

However, it can be seen as bad practice to reach into other files than the *index.js* in the module, as it breaks the rules of encapsulation.

Code Playground: src/App/index.js

```
// bad practice, don't do it
import SubmitButton from '../Buttons/SubmitButton';
```

We refactored the source code in a module with the constraints of encapsulation. Again, we'll keep things simple in this book, but you should always refactor your source code to keep it clean.

Exercises:

- Refactor your *src/App.js* file into multiple component modules when you finished the book

Snapshot Tests with Jest

Testing source code is essential to programming, and should be seen as mandatory. We want to keep the quality of your code high and make sure everything works before using it in production. We will use the testing pyramid as our guide.

The testing pyramid includes end-to-end tests, integration tests, and unit tests. A unit test is for an isolated and small block of code, such a single function. However, sometimes units work well in isolation but not in combination with other units, so they need to be tested as a group. That's where integration tests can help us figure out if units work well together. An end-to-end test is a simulation of a real-life scenario, such as the automated setup of a browser simulating the login flow in a web application. Where unit tests are fast and easy to write and maintain, end-to-end tests are at the opposite of the spectrum.

We want to have many unit tests covering the isolated functions. After that, we can use several integration tests to make sure the most important functions work in combination as expected. Finally, we may need a few end-to-end tests to simulate critical scenarios.

The foundation for testing in React are component tests, generalized partly as unit tests and partly as snapshot tests. Unit tests for our components will be covered in the next chapter with a library called Enzyme. In this section, we focus on snapshot tests, using Jest[173].

Jest is a JavaScript testing framework used by Facebook. It is used for component tests by the React community. Fortunately, *create-react-app* already comes with Jest, so you don't need to set it up.

Before you can test your first components, you have to export them from your *src/App.js* file. Afterward, you can test them in a different file.

src/App.js

```
...

class App extends Component {
  ...
}

...

export default App;

export {
  Button,
  Search,
  Table,
};
```

[173] https://jestjs.io/

In your *App.test.js* file, you will find the first test that came with *create-react-app*. It verifies the App component will render without errors.

src/App.test.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

it('renders without crashing', () => {
  const div = document.createElement('div');
  ReactDOM.render(<App />, div);
  ReactDOM.unmountComponentAtNode(div);
});
```

The "it"-block describes one test case. It comes with a test description, and when you test it, it can either succeed or fail. Furthermore, you could wrap it into a "describe"-block that defines your test suite. A test suite could include a bunch of the "it"-blocks for one specific component. You will see "describe"-blocks later. Both blocks are used to separate and organize your test cases.

Note that the `it` function is acknowledged in the JavaScript community as the function where you run a single test. However, in Jest it is often found as an alias `test` function.

You can run the test cases using the interactive *create-react-app* test script on the command line. You will get the output for all test cases on your command line interface.

Command Line

```
npm test
```

Jest enables you to write snapshot tests. These tests make a snapshot of your rendered component and runs it against future snapshots. When a future snapshot changes, you will get notified in the test. You can either accept the snapshot change, because you changed the component implementation on purpose, or deny the change and investigate for the error. It complements unit tests very well, because you only test the differences of the rendered output. It doesn't add big maintenance costs since you can accept snapshots for intentional changes.

Jest stores snapshots in a folder so it can validate the diff against a future snapshot. This also lets use share snapshots across teams when having version control such as git in place.

Before writing your first snapshot test with Jest, you have to install its utility library:

Command Line

```
npm install --save-dev react-test-renderer
```

Now you can extend the App component test with your first snapshot test. First, import the new functionality from the node package and wrap your previous "it"-block for the App component into a descriptive "describe"-block. In this case, the test suite is only for the App component.

src/App.test.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import renderer from 'react-test-renderer';
import App from './App';

describe('App', () => {

  it('renders without crashing', () => {
    const div = document.createElement('div');
    ReactDOM.render(<App />, div);
    ReactDOM.unmountComponentAtNode(div);
  });

});
```

Now you can implement your first snapshot test by using a "test"-block:

src/App.test.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import renderer from 'react-test-renderer';
import App from './App';

describe('App', () => {

  it('renders without crashing', () => {
    const div = document.createElement('div');
    ReactDOM.render(<App />, div);
    ReactDOM.unmountComponentAtNode(div);
  });

  test('has a valid snapshot', () => {
    const component = renderer.create(
```

```
      <App />
    );
    const tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });

});
```

Run your tests again and observe how they succeed or fail. Once you change the output of the render block in your App component, the snapshot test should fail. Then you can decide to update the snapshot or investigate in your App component.

The `renderer.create()` function creates a snapshot of your App component. It renders it virtually, and then stores the DOM into a snapshot. Afterward, the snapshot is expected to match the previous version from the last test. This is how we make sure the DOM stays the same and doesn't change anything by accident.

Let's add more tests for our independent components. First, the Search component:

src/App.test.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import renderer from 'react-test-renderer';
import App, { Search } from './App';

...

describe('Search', () => {

  it('renders without crashing', () => {
    const div = document.createElement('div');
    ReactDOM.render(<Search>Search</Search>, div);
    ReactDOM.unmountComponentAtNode(div);
  });

  test('has a valid snapshot', () => {
    const component = renderer.create(
      <Search>Search</Search>
    );
    const tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });

});
```

The Search component has two tests similar to the App component. The first test simply renders the Search component to the DOM and verifies that there is no error during the rendering process. If there would be an error, the test would break even though there isn't any assertion (e.g. expect, match, equal) in the test block. The second snapshot test is used to store a snapshot of the rendered component and to run it against a previous snapshot. It fails when the snapshot has changed.

Second, you can test the Button component whereas the same test rules as in the Search component apply.

src/App.test.js

```
...
import App, { Search, Button } from './App';

...

describe('Button', () => {

  it('renders without crashing', () => {
    const div = document.createElement('div');
    ReactDOM.render(<Button>Give Me More</Button>, div);
    ReactDOM.unmountComponentAtNode(div);
  });

  test('has a valid snapshot', () => {
    const component = renderer.create(
      <Button>Give Me More</Button>
    );
    const tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });

});
```

Finally, the Table component that you can pass a bunch of initial props to render it with a sample list.

src/App.test.js

```
...
import App, { Search, Button, Table } from './App';

...

describe('Table', () => {

  const props = {
    list: [
      { title: '1', author: '1', num_comments: 1, points: 2, objectID: 'y' },
      { title: '2', author: '2', num_comments: 1, points: 2, objectID: 'z' },
    ],
  };

  it('renders without crashing', () => {
    const div = document.createElement('div');
    ReactDOM.render(<Table { ...props } />, div);
  });

  test('has a valid snapshot', () => {
    const component = renderer.create(
      <Table { ...props } />
    );
    const tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });

});
```

Snapshot tests usually stay pretty basic. You only want to make sure the component doesn't change its output. Once it does, you have to decide if you accept the changes, otherwise you have to fix the component.

Exercises:

- Confirm your source code for the last section[174]
  - Confirm the changes from the last section[175]
- See how a snapshot test fails once you change your component's return value in the `render()` method

[174] http://bit.ly/2H9GkI3
[175] http://bit.ly/2Ha22f6

    - Accept or deny the snapshot change(s)
- Keep your snapshots tests up to date when the implementation of components change in next chapters
- Read about Jest in React[176]

[176] https://jestjs.io/docs/en/tutorial-react

Unit Tests with Enzyme

Enzyme[177] is a testing utility by Airbnb to assert, manipulate, and traverse React components. It is used to conduct unit tests to complement snapshot tests in React. First we have to install it along with its extension, since it doesn't come with *create-react-app*.

Command Line

```
npm install --save-dev enzyme react-addons-test-utils enzyme-adapter-react-16
```

Second, we include it in the test setup and we initialize its adapter.

src/App.test.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import renderer from 'react-test-renderer';
import Enzyme from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';
import App, { Search, Button, Table } from './App';

Enzyme.configure({ adapter: new Adapter() });
```

Now you can write your first unit test in the Table "describe"-block. You will use `shallow()` to render your component and assert that the Table was passed two items. The assertion simply checks if the element has two elements with the class `table-row`.

src/App.test.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import renderer from 'react-test-renderer';
import Enzyme, { shallow } from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';
import App, { Search, Button, Table } from './App';

...

describe('Table', () => {

  const props = {
    list: [
      { title: '1', author: '1', num_comments: 1, points: 2, objectID: 'y' },
```

[177] https://github.com/airbnb/enzyme

```
      { title: '2', author: '2', num_comments: 1, points: 2, objectID: 'z' },
    ],
  };

  ...

  it('shows two items in list', () => {
    const element = shallow(
      <Table { ...props } />
    );

    expect(element.find('.table-row').length).toBe(2);
  });

});
```

Shallow renders the component without its child components, so you can dedicate the test to one component.

Enzyme has three rendering mechanisms in its API. You already know `shallow()`, but there is also `mount()` and `render()`. Both instantiate instances of the parent component and all child components. `mount()` gives you access to the component lifecycle methods. These are the rules of them as to when to use each mechanism:

- Always begin with a shallow test
- If `componentDidMount()` or `componentDidUpdate()` should be tested, use `mount()`
- If you want to test component lifecycle and children behavior, use `mount()`
- If you want to test a component's children rendering with less overhead than `mount()` and you are not interested in lifecycle methods, use `render()`

Continue to unit test your components, but be sure to keep the tests simple and maintainable. Otherwise you will have to refactor them once you change your components. This is one of the main reasons Facebook introduced snapshot tests with Jest.

Exercises:

- Confirm your source code for the last section[178]
  - Confirm the changes from the last section[179]
- Write a unit test with Enzyme for your Button component
- Keep your unit tests up to date during the following chapters

[178] http://bit.ly/2HaT2Gl
[179] http://bit.ly/2H842oa

- Read about Enzyme and its rendering API[180]
- Read about testing React applications[181]

[180] https://github.com/airbnb/enzyme
[181] https://www.robinwieruch.de/react-testing-tutorial

Component Interface with PropTypes

TypeScript[182] and Flow[183] are used to introduce a type system to JavaScript. A typed language is less error prone because the code gets validated based on its program text. Editors and other utilities can catch these errors before the program runs.

In the book, you will not introduce Flow or TypeScript, but another useful way to check your types in components: React comes with a built-in type checker to prevent bugs. You can use PropTypes to describe your component interface. All the props passed from a parent component to a child get validated based on the PropTypes interface assigned to the child.

This section will show you to make components type safe with PropTypes. I will omit the changes for the following chapters, since they add unnecessary code refactorings, but you should update them along the way to keep your components interface type safe.

First, install a separate package for React:

Command Line

```
npm install prop-types
```

Now, you can import the PropTypes.

src/App.js

```
import React, { Component } from 'react';
import axios from 'axios';
import PropTypes from 'prop-types';
```

Let's start to assign a props interface to the components:

src/App.js

```
const Button = ({
  onClick,
  className = '',
  children,
}) =>
  <button
    onClick={onClick}
    className={className}
    type="button"
  >
    {children}
```

[182] https://www.typescriptlang.org/
[183] https://flowtype.org/

```
  </button>

Button.propTypes = {
  onClick: PropTypes.func,
  className: PropTypes.string,
  children: PropTypes.node,
};
```

Essentially, we want to take every argument from the function signature and assign a PropType to it. The basic PropTypes for primitives and complex objects are:

- PropTypes.array
- PropTypes.bool
- PropTypes.func
- PropTypes.number
- PropTypes.object
- PropTypes.string

There are two more PropTypes that can define a renderable fragment (node), e.g. a string, and a React element:

- PropTypes.node
- PropTypes.element

You already used the `node` PropType for the Button component. There are more PropType definitions in the official React documentation.

At the moment, all the defined PropTypes for the Button are optional. The parameters can be `null` or `undefined`. But for several props you want to enforce that they be defined. To do this, we make it a requirement that these props are passed to the component.

src/App.js

```
Button.propTypes = {
  onClick: PropTypes.func.isRequired,
  className: PropTypes.string,
  children: PropTypes.node.isRequired,
};
```

The `className` is not required, because it can default to an empty string. Next you will define a PropType interface for the Table component:

src/App.js

```
Table.propTypes = {
  list: PropTypes.array.isRequired,
  onDismiss: PropTypes.func.isRequired,
};
```

You can define the content of an array PropType more explicitly:

src/App.js

```
Table.propTypes = {
  list: PropTypes.arrayOf(
    PropTypes.shape({
      objectID: PropTypes.string.isRequired,
      author: PropTypes.string,
      url: PropTypes.string,
      num_comments: PropTypes.number,
      points: PropTypes.number,
    })
  ).isRequired,
  onDismiss: PropTypes.func.isRequired,
};
```

Only the `objectID` is required, because some of the code depends on it. The other properties are only displayed, so they are not required. Moreover you cannot be sure the Hacker News API always has a defined property for each object in the array.

You can also define default props in your component. The `className` property has an ES6 default parameter in the component signature:

src/App.js

```
const Button = ({
  onClick,
  className = '',
  children
}) =>
  ...
```

Replace it with the internal React default prop:

src/App.js

```
const Button = ({
  onClick,
  className,
  children
}) =>
  <button
    onClick={onClick}
    className={className}
    type="button"
  >
    {children}
  </button>

Button.defaultProps = {
  className: '',
};
```

Like the ES6 default parameter, the default prop ensures the property is set to a default value when the parent component doesn't specify it. The PropType type check happens after the default prop is evaluated.

If you run your tests again, you might see PropType errors for your components on your command line. It happens when we don't define all props for components in the tests that are required in the PropType definition. The tests themselves all pass correctly though. Make sure to pass all required props to the components in your tests to avoid these errors.

Exercises:

- Confirm your source code for the last section[184]
  - Confirm the changes from the last section[185]
  - Note that the changes get reverted for the next section
- Define the PropType interface for the Search component
- Add and update the PropType interfaces when you add and update components in the next chapters
- Read about React PropTypes[186]

[184] http://bit.ly/2H8ILul
[185] http://bit.ly/2H843se
[186] https://reactjs.org/docs/typechecking-with-proptypes.html

Debugging with React Developer Tools

React Developer Tools[187] lets you inspect the React components hierarchy, props and state. It comes as a browser extension for Chrome and Firefox and as a standalone app that works with other environments. Once installed, the extension icon will light up on websites using React. On such pages, you will see a tab called "React" in your browser's developer tools.

Let's try it on our Hacker News application. On most browsers, a quick way to bring the *dev tools* up is to right-click on the page and than hit "Inspect". Do it when your application is loaded, then click on the "React" tab. You should see its elements hierarchy, being `<App>` the root element. If you expand it, you will find instances of your `<Search>`, `<Table>` and `<Button>` components, as well.

The extension shows on the side pane the component's state and props for the selected element. For instance, if you click on `<App>`, you will see that it has no props, but it already has a state. A very straightforward debugging technique is monitoring your application's state changes from user interaction.

First, check the "Highlight Updates" option, usually above the elements tree. Second, type a different search term in the application's input field. Only `searchTerm` will be changed in the component's state. We already knew that would happen, but now we can see it working as planned.

Finally, press the "Search" button. The `searchKey` state immediately changes to same value as `searchTerm`, and then the response object is added to `results`. The asynchronous nature of your code is now visible.

If you right-click on any element, a dropdown menu will show several useful options. For instance, you could copy the element's props or name, find the corresponding DOM node, or jump to the application's source code in the browser. The last option is very useful for inserting breakpoints and debugging your JavaScript functions.

Exercises:

- Install the React Developer Tools[188] extension on your favorite browser
  - Run your Hacker News Clone application and inspect it using the extension
  - Experiment with state and props changes
  - Watch what happens when you trigger an asynchronous request
  - Perform several requests, including repeated ones. Watch the cache mechanism working
- Read about how to debug your JavaScript functions in the browser[189]
- Read about (and use) the React Profiler[190]

You have learned how to organize and test React code! Let's recap the last chapter:

[187] https://github.com/facebook/react-devtools
[188] https://github.com/facebook/react-devtools
[189] https://developers.google.com/web/tools/chrome-devtools/javascript/
[190] https://reactjs.org/blog/2018/09/10/introducing-the-react-profiler.html

- **React**
  - PropTypes let you define type checks for components
  - Jest allows you to write snapshot tests for your components
  - Enzyme allows you to write unit tests for your components
  - React Developer Tools is a helpful debugging tool
- **ES6**
  - import and export statements help you to organize your code
- **General**
  - code organization allows you to scale your application with best practices

You can find the source code in the official repository[191].

[191] http://bit.ly/2H647bK

Advanced React Components

This chapter focuses on implementing advanced React components. You will learn how to implement higher-order components, and we will dive into more advanced topics in React.

Ref a DOM Element

Sometimes you need to interact with your DOM nodes in React. The `ref` attribute gives you access to a node in your elements. That is usually an anti pattern in React, because you should use its declarative way of doing things and its unidirectional data flow. We learned about it when we introduced the first search input field, but there are certain cases where you need access to the DOM node. The official documentation mentions three:

- to use the DOM API (focus, media playback etc.)
- to invoke imperative DOM node animations
- to integrate with a third-party library that needs the DOM node (e.g. D3.js[192])

We'll use the Search component as an example. When the application renders for the first time, the input field should be focused. This is one case where we need access to the DOM API. The `ref` attribute can be used in both functional stateless components and ES6 class components. In this example, we need a lifecycle method, so the approach is showcased using the `ref` attribute with an ES6 class component.

The initial step is to refactor the functional stateless component to an ES6 class component.

src/App.js

```
class Search extends Component {
  render() {
    const {
      value,
      onChange,
      onSubmit,
      children
    } = this.props;

    return (
      <form onSubmit={onSubmit}>
        <input
          type="text"
          value={value}
          onChange={onChange}
        />
        <button type="submit">
          {children}
        </button>
      </form>
```

[192] https://d3js.org/

```
    );
  }
}
```

The `this` object of an ES6 class component helps us to reference the DOM element with the `ref` attribute.

src/App.js

```
class Search extends Component {
  render() {
    const {
      value,
      onChange,
      onSubmit,
      children
    } = this.props;

    return (
      <form onSubmit={onSubmit}>
        <input
          type="text"
          value={value}
          onChange={onChange}
          ref={el => this.input = el}
        />
        <button type="submit">
          {children}
        </button>
      </form>
    );
  }
}
```

Now you can focus the input field when the component mounted by using the `this` object, the appropriate lifecycle method, and the DOM API.

src/App.js

```
class Search extends Component {
  componentDidMount() {
    if (this.input) {
      this.input.focus();
    }
  }

  render() {
    const {
      value,
      onChange,
      onSubmit,
      children
    } = this.props;

    return (
      <form onSubmit={onSubmit}>
        <input
          type="text"
          value={value}
          onChange={onChange}
          ref={el => this.input = el}
        />
        <button type="submit">
          {children}
        </button>
      </form>
    );
  }
}
```

The input field should be focused when the application renders. We access to the `ref` in a functional stateless component without the `this` object using the following functional stateless component:

src/App.js

```
const Search = ({
  value,
  onChange,
  onSubmit,
  children
}) => {
  let input;
  return (
    <form onSubmit={onSubmit}>
      <input
        type="text"
        value={value}
        onChange={onChange}
        ref={el => this.input = el}
      />
      <button type="submit">
        {children}
      </button>
    </form>
  );
}
```

Now we can access the input DOM element. In our case it wouldn't help much since there's no lifecycle method in a functional stateless component to trigger the focus. So we are not going to make use of the input variable here. In the future, though, you may encounter cases where it makes sense to use a functional stateless component with the `ref` attribute.

Exercises

- Confirm your source code for the last section[193]
  - Confirm the changes from the last section[194]
  - Note that the changes get reverted for the next section
- Read about the usage of the ref attribute in React[195]
- Read about the ref attribute in general in React[196]

[193] http://bit.ly/2H8597l
[194] http://bit.ly/2H8EoQ4
[195] https://www.robinwieruch.de/react-ref-attribute-dom-node/
[196] https://reactjs.org/docs/refs-and-the-dom.html

Loading ...

Now we get back to the application, where we'll show a loading indicator when a search request submits to the Hacker News API. The request is asynchronous, so you should show your user feedback that something is happening. Let's define a reusable Loading component in your *src/App.js* file.

src/App.js

```
const Loading = () =>
  <div>Loading ...</div>
```

Now we need a property to store the loading state. Based on the loading state, we can decide to show the Loading component later.

src/App.js

```
class App extends Component {
  constructor(props) {
    super(props);

    this.state = {
      results: null,
      searchKey: '',
      searchTerm: DEFAULT_QUERY,
      error: null,
      isLoading: false,
    };

    ...
  }

  ...

}
```

The initial value of that `isLoading` property is false. We don't load anything before the App component is mounted. When the request is made, the loading state is set to true. The request will succeed eventually, and you can set the loading state to false.

src/App.js

```
class App extends Component {

  ...

  setSearchTopStories(result) {
    ...

    this.setState({
      results: {
        ...results,
        [searchKey]: { hits: updatedHits, page }
      },
      isLoading: false
    });
  }

  fetchSearchTopStories(searchTerm, page = 0) {
    this.setState({ isLoading: true });

    axios(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}${pa\
ge}&${PARAM_HPP}${DEFAULT_HPP}`)
      .then(result => this.setSearchTopStories(result.data))
      .catch(error => this.setState({ error }));
  }

  ...

}
```

In the last step, we used the Loading component in the App component. A conditional rendering based on the loading state will decide whether to show a Loading component or the Button component. The latter is the button to fetch more data.

src/App.js

```
class App extends Component {

  ...

  render() {
    const {
      searchTerm,
      results,
      searchKey,
      error,
      isLoading
    } = this.state;

    ...

    return (
      <div className="page">
        ...
        <div className="interactions">
          { isLoading
            ? <Loading />
            : <Button
                onClick={() => this.fetchSearchTopStories(searchKey, page + 1)}
              >
              More
            </Button>
          }
        </div>
      </div>
    );
  }
}
```

Initially, the Loading component will show when you start your application, because you make a request on `componentDidMount()`. There is no Table component, because the list is empty. When the response returns from the Hacker News API, the result is shown, the loading state is set to false, and the Loading component disappears. Instead, the "More" button to fetch more data appears. Once you fetch more data, the button will disappear again and the Loading component will appear.

Exercises:

- Confirm your source code for the last section[197]
  - Confirm the changes from the last section[198]
- Use a library such as Font Awesome[199] to show a loading icon instead of the "Loading ..." text

[197] http://bit.ly/2HkQk0k
[198] http://bit.ly/2Hns4uz
[199] https://fontawesome.com/

Higher-Order Components

Higher-order components (HOC) are an advanced concept in React. HOCs are an equivalent to higher-order functions. They take any input, usually a component, but also optional arguments, and return a component as output. The returned component is an enhanced version of the input component, and it can be used in your JSX.

HOCs are used for different use cases. They can prepare properties, manage state, or alter the representation of a component. One case is to use a HOC as a helper for a conditional rendering. Imagine you have a List component that renders a list of items or nothing, because the list is empty or null. The HOC could shield away that the list would render nothing when there is no list. On the other hand, the plain List component doesn't need to bother anymore about a non existent list, as it only cares about rendering the list.

Let's do a simple HOC that takes a component as input and returns a component. You can place it in your *src/App.js* file.

src/App.js

```
function withFeature(Component) {
  return function(props) {
    return <Component { ...props } />;
  }
}
```

It is a useful convention to prefix a HOC with `with`. Since you are using JavaScript ES6, you can express the HOC better with an ES6 arrow function.

src/App.js

```
const withEnhancement = (Component) => (props) =>
  <Component { ...props } />
```

In our example, the input component stays the same as the output, so nothing happens. The output component should show the Loading component when the loading state is true, otherwise it should show the input component. A conditional rendering is a great use case for an HOC.

src/App.js

```
const withLoading = (Component) => (props) =>
  props.isLoading
    ? <Loading />
    : <Component { ...props } />
```

Based on the loading property, you can apply a conditional rendering. The function will return the Loading component or the input component. In general, it can be very efficient to spread an object like the props object in the previous example as input for a component. See the difference in the following code snippet:

Code Playground

```
// before you would have to destructure the props before passing them
const { firstname, lastname } = props;
<UserProfile firstname={firstname} lastname={lastname} />

// but you can use the object spread operator to pass all object properties
<UserProfile { ...props } />
```

We passed all the props including the `isLoading` property by spreading the object into the input component. The input component may not care about the `isLoading` property. You can use the ES6 rest destructuring to avoid it:

src/App.js

```
const withLoading = (Component) => ({ isLoading, ...rest }) =>
  isLoading
    ? <Loading />
    : <Component { ...rest } />
```

It takes one property out of the object, but keeps the remaining object, which also works with multiple properties. More can be read about it in Mozilla's destructuring assignment[200].

Now you can use the HOC in JSX. Maybe you want to show either the "More" button or the Loading component. The Loading component is already encapsulated in the HOC, but an input component is missing. For showing either a Button component or a Loading component, the Button is the input component of the HOC. The enhanced output component is a ButtonWithLoading component.

src/App.js

```
const Button = ({
  onClick,
  className = '',
  children,
}) =>
  <button
    onClick={onClick}
    className={className}
    type="button"
```

[200] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

```
  >
    {children}
  </button>

const Loading = () =>
  <div>Loading ...</div>

const withLoading = (Component) => ({ isLoading, ...rest }) =>
  isLoading
    ? <Loading />
    : <Component { ...rest } />

const ButtonWithLoading = withLoading(Button);
```

Everything is defined now. As a last step, you have to use the ButtonWithLoading component, which receives the loading state as an additional property. While the HOC consumes the loading property, all other props get passed to the Button component.

src/App.js

```
class App extends Component {

  ...

  render() {
    ...
    return (
      <div className="page">
        ...
        <div className="interactions">
          <ButtonWithLoading
            isLoading={isLoading}
            onClick={() => this.fetchSearchTopStories(searchKey, page + 1)}
          >
            More
          </ButtonWithLoading>
        </div>
      </div>
    );
  }
}
```

Note that when you run your tests again, your snapshot test for the App component fails. The diff might look like the following on the command line:

Command Line

```
-     <button
-       className=""
-       onClick={[Function]}
-       type="button"
-     >
-       More
-     </button>
+     <div>
+       Loading ...
+     </div>
```

You can either fix the component now, when you think there is something wrong about it, or can accept the new snapshot of it. Because you introduced the Loading component in this chapter, you can accept the altered snapshot test on the command line in the interactive test.

Higher-order components are an advanced pattern in React. They have multiple purposes: improved reusability of components, greater abstraction, composability of components, and manipulations of props, state and view. I encourage you to read gentle introduction to higher-order components[201]. It gives you another approach to learn them, shows you an elegant way to use them in a functional programming way, and solves the problem of conditional rendering with higher-order components.

Exercises:

- Confirm your source code for the last section[202]
  - Confirm the changes from the last section[203]
- Read a gentle introduction to higher-order components[204]
- Experiment with the HOC you have created
- Think about a use case where another HOC would make sense
  - Implement the HOC, if there is a use case

[201] https://www.robinwieruch.de/gentle-introduction-higher-order-components/
[202] http://bit.ly/2HkoP7c
[203] http://bit.ly/2Ho9uT9
[204] https://www.robinwieruch.de/gentle-introduction-higher-order-components/

Advanced Sorting

We implemented a client and server-side search interaction earlier. Since you have a Table component, it makes sense to enhance it with advanced interactions. Next, we'll introduce a sort functionality for each column by using the column headers of the Table.

It is possible to write your own sort function, but I prefer to use a utility library like Lodash[205] for these cases. There are other options, but we'll install Lodash for our sort function:

Command Line

```
npm install lodash
```

Now we import the sort functionality of Lodash in your *src/App.js* file:

src/App.js

```
import React, { Component } from 'react';
import axios from 'axios';
import { sortBy } from 'lodash';
import './App.css';
```

Now we have several columns in Table: title, author, comments and points columns. You can define sort functions where each takes a list and returns a list of items sorted by a specific property. Additionally, you will need a default sort function that doesn't sort, but returns the unsorted list. This will be the initial state.

src/App.js

```
...

const SORTS = {
  NONE: list => list,
  TITLE: list => sortBy(list, 'title'),
  AUTHOR: list => sortBy(list, 'author'),
  COMMENTS: list => sortBy(list, 'num_comments').reverse(),
  POINTS: list => sortBy(list, 'points').reverse(),
};

class App extends Component {
  ...
}
...
```

[205] https://lodash.com/

Two of the sort functions return a reversed list. That's to see the items with the highest comments and points, rather than the items with the lowest counts when the list is sorted for the first time.

The `SORTS` object allows you to reference any sort function now.

Again, the App component is responsible for storing the state of the sort. The initial state will be the default sort function, which doesn't sort at all and returns the input list as output.

src/App.js

```
this.state = {
  results: null,
  searchKey: '',
  searchTerm: DEFAULT_QUERY,
  error: null,
  isLoading: false,
  sortKey: 'NONE',
};
```

Once we choose a different `sortKey`, like the `AUTHOR` key, we sort the list with the appropriate sort function from the `SORTS` object.

Now we define a new class method in App component that sets a `sortKey` to the local component state, then `sortKey` can be used to retrieve the sorting function to apply it to the list:

src/App.js

```
class App extends Component {
  constructor(props) {

    ...

    this.needsToSearchTopStories = this.needsToSearchTopStories.bind(this);
    this.setSearchTopStories = this.setSearchTopStories.bind(this);
    this.fetchSearchTopStories = this.fetchSearchTopStories.bind(this);
    this.onSearchSubmit = this.onSearchSubmit.bind(this);
    this.onSearchChange = this.onSearchChange.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
    this.onSort = this.onSort.bind(this);
  }

  ...

  onSort(sortKey) {
    this.setState({ sortKey });
  }
```

```
  ...

}
```

The next step is to pass the method and `sortKey` to the Table component.

src/App.js

```
class App extends Component {

  ...

  render() {
    const {
      searchTerm,
      results,
      searchKey,
      error,
      isLoading,
      sortKey
    } = this.state;

    ...

    return (
      <div className="page">
        ...
        <Table
          list={list}
          sortKey={sortKey}
          onSort={this.onSort}
          onDismiss={this.onDismiss}
        />
        ...
      </div>
    );
  }
}
```

The Table component is responsible for sorting the list. It takes one of the `SORT` functions by `sortKey` and passes the list as input, after which it keeps mapping over the sorted list.

src/App.js

```
const Table = ({
  list,
  sortKey,
  onSort,
  onDismiss
}) =>
  <div className="table">
    {SORTS[sortKey](list).map(item =>
      <div key={item.objectID} className="table-row">
        ...
      </div>
    )}
  </div>
```

In theory, the list should get sorted by one of the functions. But the default sort is set to `NONE`, so nothing is sorted yet, as nothing executes the `onSort()` method to change the `sortKey`. We extend the Table with a row of column headers that use Sort components in columns to sort each column:

src/App.js

```
const Table = ({
  list,
  sortKey,
  onSort,
  onDismiss
}) =>
  <div className="table">
    <div className="table-header">
      <span style={{ width: '40%' }}>
        <Sort
          sortKey={'TITLE'}
          onSort={onSort}
        >
          Title
        </Sort>
      </span>
      <span style={{ width: '30%' }}>
        <Sort
          sortKey={'AUTHOR'}
          onSort={onSort}
        >
          Author
```

```
        </Sort>
      </span>
      <span style={{ width: '10%' }}>
        <Sort
          sortKey={'COMMENTS'}
          onSort={onSort}
        >
          Comments
        </Sort>
      </span>
      <span style={{ width: '10%' }}>
        <Sort
          sortKey={'POINTS'}
          onSort={onSort}
        >
          Points
        </Sort>
      </span>
      <span style={{ width: '10%' }}>
        Archive
      </span>
    </div>
    {SORTS[sortKey](list).map(item =>
      ...
    )}
  </div>
```

Each Sort component gets a specific `sortKey` and the general `onSort()` function. Internally, it calls the method with the `sortKey` to set the specific key.

src/App.js

```
const Sort = ({ sortKey, onSort, children }) =>
  <Button onClick={() => onSort(sortKey)}>
    {children}
  </Button>
```

As you can see, the Sort component reuses your common Button component. On a button click, each individual passed `sortKey` is set by the `onSort()` method, so the list is sorted when column headers are selected.

Now we'll improve the look of the button in the column header. Let's give it a proper `className`:

src/App.js

```
const Sort = ({ sortKey, onSort, children }) =>
  <Button
    onClick={() => onSort(sortKey)}
    className="button-inline"
  >
    {children}
  </Button>
```

This was done to improve the UI. The next goal is to implement a reverse sort. The list should perform a reverse sort once you click a Sort component twice. First, you need to define the reverse state with a boolean. The sort can be either reversed or non-reversed.

src/App.js

```
this.state = {
  results: null,
  searchKey: '',
  searchTerm: DEFAULT_QUERY,
  error: null,
  isLoading: false,
  sortKey: 'NONE',
  isSortReverse: false,
};
```

Now in your sort method, you can evaluate if the list is reverse sorted. It is reverse if the `sortKey` in the state is the same as the incoming `sortKey` and the reverse state is not already set to true.

src/App.js

```
onSort(sortKey) {
  const isSortReverse = this.state.sortKey === sortKey && !this.state.isSortReverse;
  this.setState({ sortKey, isSortReverse });
}
```

Again, we pass the reverse prop to your Table component:

src/App.js

```
class App extends Component {

  ...

  render() {
    const {
      searchTerm,
      results,
      searchKey,
      error,
      isLoading,
      sortKey,
      isSortReverse
    } = this.state;

    ...

    return (
      <div className="page">
        ...
        <Table
          list={list}
          sortKey={sortKey}
          isSortReverse={isSortReverse}
          onSort={this.onSort}
          onDismiss={this.onDismiss}
        />
        ...
      </div>
    );
  }
}
```

The Table has to have an arrow function block body to compute the data now:

src/App.js

```
const Table = ({
  list,
  sortKey,
  isSortReverse,
  onSort,
  onDismiss
}) => {
  const sortedList = SORTS[sortKey](list);
  const reverseSortedList = isSortReverse
    ? sortedList.reverse()
    : sortedList;

  return(
    <div className="table">
      <div className="table-header">
        ...
      </div>
      {reverseSortedList.map(item =>
        ...
      )}
    </div>
  );
}
```

Finally, we want to give the user visual feedback to distinguish which column is actively sorted. Each Sort component has its specific `sortKey` already, which can be used to identify the activated sort. We pass the `sortKey` from the internal component state as active sort key to your Sort component:

src/App.js

```
const Table = ({
  list,
  sortKey,
  isSortReverse,
  onSort,
  onDismiss
}) => {
  const sortedList = SORTS[sortKey](list);
  const reverseSortedList = isSortReverse
    ? sortedList.reverse()
    : sortedList;
```

```
  return(
    <div className="table">
      <div className="table-header">
        <span style={{ width: '40%' }}>
          <Sort
            sortKey={'TITLE'}
            onSort={onSort}
            activeSortKey={sortKey}
          >
            Title
          </Sort>
        </span>
        <span style={{ width: '30%' }}>
          <Sort
            sortKey={'AUTHOR'}
            onSort={onSort}
            activeSortKey={sortKey}
          >
            Author
          </Sort>
        </span>
        <span style={{ width: '10%' }}>
          <Sort
            sortKey={'COMMENTS'}
            onSort={onSort}
            activeSortKey={sortKey}
          >
            Comments
          </Sort>
        </span>
        <span style={{ width: '10%' }}>
          <Sort
            sortKey={'POINTS'}
            onSort={onSort}
            activeSortKey={sortKey}
          >
            Points
          </Sort>
        </span>
        <span style={{ width: '10%' }}>
          Archive
        </span>
      </div>
```

```
      {reverseSortedList.map(item =>
          ...
      )}
    </div>
  );
}
```

Now the user will know whether sort is active based on the `sortKey` and `activeSortKey` . Give your Sort component an extra `className` attribute, in case it is sorted, to give visual feedback:

src/App.js

```
const Sort = ({
  sortKey,
  activeSortKey,
  onSort,
  children
}) => {
  const sortClass = ['button-inline'];

  if (sortKey === activeSortKey) {
    sortClass.push('button-active');
  }

  return (
    <Button
      onClick={() => onSort(sortKey)}
      className={sortClass.join(' ')}
    >
      {children}
    </Button>
  );
}
```

We can define `sortClass` more efficiently using a library called classnames, which is installed using npm:

Command Line

```
npm install classnames
```

After installation, we import it on top of the *src/App.js* file.

src/App.js

```
import React, { Component } from 'react';
import axios from 'axios';
import { sortBy } from 'lodash';
import classNames from 'classnames';
import './App.css';
```

Now we can define `className` with conditional classes:

src/App.js

```
const Sort = ({
  sortKey,
  activeSortKey,
  onSort,
  children
}) => {
  const sortClass = classNames(
    'button-inline',
    { 'button-active': sortKey === activeSortKey }
  );

  return (
    <Button
      onClick={() => onSort(sortKey)}
      className={sortClass}
    >
      {children}
    </Button>
  );
}
```

There will be failing snapshot and unit tests for the Table component. Since we intentionally changed again our component representations, we accept the snapshot tests, but we still need to fix the unit test. In *src/App.test.js* , provide a `sortKey` and the `isSortReverse` boolean for the Table component.

src/App.test.js

```
...

describe('Table', () => {

  const props = {
    list: [
      { title: '1', author: '1', num_comments: 1, points: 2, objectID: 'y' },
      { title: '2', author: '2', num_comments: 1, points: 2, objectID: 'z' },
    ],
    sortKey: 'TITLE',
    isSortReverse: false,
  };

  ...

});
```

We may need to accept the failing snapshot tests again for the Table component, because we provided extended props for it. The advanced sort interaction is finally complete.

Exercises:

- Confirm your source code for the last section[206]
  - Confirm the changes from the last section[207]
- Use a library like Font Awesome[208] to indicate the (reverse) sort. It could be an arrow up or arrow down icon next to each Sort header
- Read about the classnames library[209]

[206] http://bit.ly/2HteRAe
[207] http://bit.ly/2HnYFAh
[208] https://fontawesome.com/
[209] https://github.com/JedWatson/classnames

You have learned advanced component techniques in React! Let's recap the chapter:

- **React**
  - The `ref` attribute to reference DOM elements
  - Higher-order components are a common way to build advanced components
  - Implementation of advanced interactions in React
  - Conditional classNames with a neat helper library
- **ES6**
  - Rest destructuring to split up objects and arrays

You can always find the source code in the official repository[210].

[210] http://bit.ly/2HteRAe

State Management in React

We have already covered the basics of state management in React in the previous chapters by using React's local state, so this chapter will dig a bit deeper. It will expand on the best practices, how to apply them, and why you could consider using a third-party state management library.

Lifting State

Only the App component is a stateful ES6 component in your application. It handles a lot of application state and logic in its class methods. Moreover, we pass a lot of properties to the Table component, most of which are only used in there. It's not important that the App component knows about them, so the sort functionality could be moved into the Table component.

Moving substate from one component to another is known as *lifting state*. We want to move state that isn't used in the App component into the Table component, down from parent to child component. To deal with state and class methods in the Table component, it has to become an ES6 class component. The refactoring from functional stateless component to ES6 class component is straightforward.

Your Table component as a functional stateless component:

src/App.js

```
const Table = ({
  list,
  sortKey,
  isSortReverse,
  onSort,
  onDismiss
}) => {
  const sortedList = SORTS[sortKey](list);
  const reverseSortedList = isSortReverse
    ? sortedList.reverse()
    : sortedList;

  return(
    ...
  );
}
```

Your Table component as an ES6 class component:

src/App.js

```
class Table extends Component {
  render() {
    const {
      list,
      sortKey,
      isSortReverse,
      onSort,
      onDismiss
    } = this.props;

    const sortedList = SORTS[sortKey](list);
    const reverseSortedList = isSortReverse
      ? sortedList.reverse()
      : sortedList;

    return (
      ...
    );
  }
}
```

Since you want to deal with state and methods in your component, you have to add a constructor and initial state.

src/App.js

```
class Table extends Component {
  constructor(props) {
    super(props);

    this.state = {};
  }

  render() {
    ...
  }
}
```

Now you can move state and class methods with the sort functionality from your App component down to your Table component.

src/App.js

```
class Table extends Component {
  constructor(props) {
    super(props);

    this.state = {
      sortKey: 'NONE',
      isSortReverse: false,
    };

    this.onSort = this.onSort.bind(this);
  }

  onSort(sortKey) {
    const isSortReverse = this.state.sortKey === sortKey &&
      !this.state.isSortReverse;

    this.setState({ sortKey, isSortReverse });
  }

  render() {
    ...
  }
}
```

Remember to remove the moved state and `onSort()` class method from your App component.

src/App.js

```
class App extends Component {
  constructor(props) {
    super(props);

    this.state = {
      results: null,
      searchKey: '',
      searchTerm: DEFAULT_QUERY,
      error: null,
      isLoading: false,
    };

    this.setSearchTopStories = this.setSearchTopStories.bind(this);
    this.fetchSearchTopStories = this.fetchSearchTopStories.bind(this);
```

```
    this.onDismiss = this.onDismiss.bind(this);
    this.onSearchSubmit = this.onSearchSubmit.bind(this);
    this.onSearchChange = this.onSearchChange.bind(this);
    this.needsToSearchTopStories = this.needsToSearchTopStories.bind(this);
  }

  ...

}
```

You can also make the Table component more lightweight. To do this, we move props that are passed to it from the App component, because they are handled internally in the Table component.

src/App.js

```
class App extends Component {

  ...

  render() {
    const {
      searchTerm,
      results,
      searchKey,
      error,
      isLoading
    } = this.state;

    ...

    return (
      <div className="page">
        ...
        { error
          ? <div className="interactions">
            <p>Something went wrong.</p>
          </div>
          : <Table
            list={list}
            onDismiss={this.onDismiss}
          />
        }
        ...
      </div>
```

```
    );
  }
}
```

In the Table component, use the internal `onSort()` method and the internal Table state:

src/App.js

```
class Table extends Component {

  ...

  render() {
    const {
      list,
      onDismiss
    } = this.props;

    const {
      sortKey,
      isSortReverse,
    } = this.state;

    const sortedList = SORTS[sortKey](list);
    const reverseSortedList = isSortReverse
      ? sortedList.reverse()
      : sortedList;

    return(
      <div className="table">
        <div className="table-header">
          <span style={{ width: '40%' }}>
            <Sort
              sortKey={'TITLE'}
              onSort={this.onSort}
              activeSortKey={sortKey}
            >
              Title
            </Sort>
          </span>
          <span style={{ width: '30%' }}>
            <Sort
              sortKey={'AUTHOR'}
              onSort={this.onSort}
```

```
              activeSortKey={sortKey}
            >
              Author
            </Sort>
          </span>
          <span style={{ width: '10%' }}>
            <Sort
              sortKey={'COMMENTS'}
              onSort={this.onSort}
              activeSortKey={sortKey}
            >
              Comments
            </Sort>
          </span>
          <span style={{ width: '10%' }}>
            <Sort
              sortKey={'POINTS'}
              onSort={this.onSort}
              activeSortKey={sortKey}
            >
              Points
            </Sort>
          </span>
          <span style={{ width: '10%' }}>
            Archive
          </span>
        </div>
        { reverseSortedList.map((item) =>
          ...
        )}
      </div>
    );
  }
}
```

We made a crucial refactoring by moving functionality and state closer into another component, and other components got more lightweight. Again, the component API of the Table got lighter because it deals internally with the sort functionality.

Lifting state can go the other way as well: from child to parent component. It is called as lifting state up. Imagine you were dealing with local state in a child component, and you want to fulfil a requirement to show the state in your parent component as well. You would have to lift up the state to your parent component. Moreover, imagine you want to show the state in a sibling component

of your child component. Again, you would lift the state up to your parent component. The parent component deals with the local state, but exposes it to both child components.

Exercises:

- Confirm your source code for the last section[211]
  - Confirm the changes from the last section[212]
- Read about lifting state in React[213]
- Read about lifting state in learn React before using Redux[214]

[211] http://bit.ly/2Ho9AtZ
[212] http://bit.ly/2Hnuh9d
[213] https://reactjs.org/docs/lifting-state-up.html
[214] https://www.robinwieruch.de/learn-react-before-using-redux/

Revisited: setState()

So far, we have used React `setState()` to manage your internal component state. We can pass an object to the function where it updates partially the local state.

Code Playground

```
this.setState({ value: 'hello'});
```

But `setState()` doesn't take only an object. In its second version, you can pass a function to update the state.

Code Playground

```
this.setState((prevState, props) => {
  ...
});
```

There is one crucial case where it makes sense to use a function over an object: when you update the state depending on the previous state or props. If you don't use a function, the local state management can cause bugs. The React `setState()` method is asynchronous. React batches `setState()` calls and executes them eventually. Sometimes, the previous state or props changes between before we can rely on it in our `setState()` call.

Code Playground

```
const { oneCount } = this.state;
const { anotherCount } = this.props;
this.setState({ count: oneCount + anotherCount });
```

Imagine that `oneCount` and `anotherCount`, thus the state or the props, change somewhere else asynchronously when you call `setState()`. In a growing application, you have more than one `setState()` call across your application. Since `setState()` executes asynchronously, you could rely in the example on stale values.

With the function approach, the function in `setState()` is a callback that operates on the state and props at the time of executing the callback function. Even though `setState()` is asynchronous, with a function it takes the state and props at the time when it is executed.

Code Playground

```
this.setState((prevState, props) => {
  const { oneCount } = prevState;
  const { anotherCount } = props;
  return { count: oneCount + anotherCount };
});
```

In our code, the `setSearchTopStories()` method relies on the previous state, and this is a good example to use a function over an object in `setState()`. Right now, it looks like the following code:

src/App.js

```
setSearchTopStories(result) {
  const { hits, page } = result;
  const { searchKey, results } = this.state;

  const oldHits = results && results[searchKey]
    ? results[searchKey].hits
    : [];

  const updatedHits = [
    ...oldHits,
    ...hits
  ];

  this.setState({
    results: {
      ...results,
      [searchKey]: { hits: updatedHits, page }
    },
    isLoading: false
  });
}
```

Here, we extracted values from the state, but updated the state depending on the previous state asynchronously. Now we'll use the functional approach to prevent bugs from a stale state:

src/App.js

```
setSearchTopStories(result) {
  const { hits, page } = result;

  this.setState(prevState => {
    ...
  });
}
```

We can move the whole block we implemented into the function by directing it to operate on the `prevState` instead of the `this.state`.

src/App.js

```
setSearchTopStories(result) {
  const { hits, page } = result;

  this.setState(prevState => {
    const { searchKey, results } = prevState;

    const oldHits = results && results[searchKey]
      ? results[searchKey].hits
      : [];

    const updatedHits = [
      ...oldHits,
      ...hits
    ];

    return {
      results: {
        ...results,
        [searchKey]: { hits: updatedHits, page }
      },
      isLoading: false
    };
  });
}
```

That will fix the issue with a stale state, but there is still one more improvement. Since it is a function, you can extract the function for improved readability. One more advantage to use a function over an object is that function can live outside of the component. We still have to use a higher-order function to pass the result to it since we want to update the state based on the fetched result from the API.

src/App.js

```
setSearchTopStories(result) {
  const { hits, page } = result;
  this.setState(updateSearchTopStoriesState(hits, page));
}
```

The `updateSearchTopStoriesState()` function has to return a function. It is a higher-order function that can be defined outside the App component. Note how the function signature changes slightly now.

src/App.js

```
const updateSearchTopStoriesState = (hits, page) => (prevState) => {
  const { searchKey, results } = prevState;

  const oldHits = results && results[searchKey]
    ? results[searchKey].hits
    : [];

  const updatedHits = [
    ...oldHits,
    ...hits
  ];

  return {
    results: {
      ...results,
      [searchKey]: { hits: updatedHits, page }
    },
    isLoading: false
  };
};

class App extends Component {
  ...
}
```

The function instead of object approach in `setState()` fixes potential bugs, while increasing the readability and maintainability of your code. Further, it becomes testable outside of the App component. I advise exporting and testing it as practice.

Exercises:

- Confirm your source code for the last section[215]
  - Confirm the changes from the last section[216]
- Read about React using state correctly[217]
- Export updateSearchTopStoriesState from the file
  - Write a test for it which passes the a payload (hits, page) and a made up previous state and finally expect a new state
- Refactor your `setState()` methods to use a function, but only when it makes sense, because it relies on props or state
- Run your tests again and verify that everything is up to date

[215] http://bit.ly/2HteTbk
[216] http://bit.ly/2Hmbz1T
[217] https://reactjs.org/docs/state-and-lifecycle.html#using-state-correctly

Taming the State

Previous chapters have shown you that state management can be a crucial topic in larger applications, as React and a lot of other SPA frameworks struggle with it. As applications get more complex, the big challenge in web applications is to tame and control the state.

Compared to other solutions, React has already taken a big step forward. A unidirectional data flow and a simple API to manage state in components is indispensable. These concepts make it easier to reason about your state and your state changes. It also makes it easier to reason about it on a component level and on an application level to a certain degree.

It is possible to introduce bugs by operating on stale state when using an object over a function in `setState()`. We lift state around to share or hide necessary state across components. Sometimes a component needs to lift up state, because its sibling component depends on it. Perhaps the component is far away in the component tree, so the stated needs to be shared across the whole component tree. Components are more involved in state management, as the main responsibility of components is representing the UI.

Because of this, there are standalone solutions to take care of state management. Libraries like Redux[218] or MobX[219] are both feasible solutions in a React application. They come with extensions, react-redux[220] and mobx-react[221], to integrate them into the React view layer. Redux and MobX are outside of the scope of this book, but I encourage you to study the different ways to handle scaling state management as your React applications become more complex.

Exercises:

- Read about external state management and how to learn it[222]
- Check out my second book about state management in React called Taming the State in React[223]

[218] https://redux.js.org/introduction
[219] https://mobx.js.org/
[220] https://github.com/reactjs/react-redux
[221] https://github.com/mobxjs/mobx-react
[222] https://www.robinwieruch.de/redux-mobx-confusion/
[223] https://roadtoreact.com/

You have learned advanced state management in React! Let's recap the last chapter:

- **React**
  - Lift state management up and down to suitable components
  - `setState()` can use a function to prevent stale state bugs
  - Existing external solutions that help you to tame the state

You can find the source code in the official repository[224].

[224] http://bit.ly/2HteTbk

Final Steps to Production

The last chapters will show you how to deploy your application to production. For this, we use the free hosting service Heroku. On the way to deploying the application, we will uncover more about *create-react-app*.

Eject

The following knowledge is not necessary to deploy your application to production, but it still bears mentioning. *create-react-app* comes with one feature to keep it extendable, but also to prevent a vendor lock-in. A vendor lock-in usually happens when you buy into a technology but there is no escape hatch from using it in the future. Fortunately, in *create-react-app* you have such an escape hatch with "eject".

In your *package.json* you will find the scripts to *start*, *test*, and *build* your application. The last script is *eject*. However, it is important you know that **it is a one-way operation**. That means **once you eject**, **you can't go back!** It is advisable to stay in the safe environment of *create-react-app* if you have just started creating applications in React.

If you feel comfortable enough to run `npm run eject`, the command copies all the configuration and dependencies to your *package.json* and a new *config/* folder. It converts the whole project into a custom setup with tooling that includes Babel and Webpack, and grants full control over all these tools.

Its official documentation says *create-react-app* is suitable for small to mid size projects, so you shouldn't feel obligated to use the "eject" command until you're ready.

Exercises:

- Read about eject[225]

[225] https://github.com/facebook/create-react-app/blob/master/packages/react-scripts/template/README.md#npm-run-eject

Deploy your App

No application should stay on localhost; eventually, it has to go live to see how it will perform in real-life scenarios. Heroku is a platform as a service where you can host your application, and it offers seamless integration with React. Specifically, it's possible to deploy a *create-react-app* in minutes, with a a zero-configuration deployment which follows the philosophy of *create-react-app*.

There are two requirements before an application can be deployed to Heroku:

- Install the Heroku CLI[226]
- Create a free Heroku account[227]

If you have installed Homebrew, you can install the Heroku CLI from command line:

Command Line

```
brew update
brew install heroku-toolbelt
```

Now you can use git and Heroku CLI to deploy your application:

Command Line

```
git init
heroku create -b https://github.com/mars/create-react-app-buildpack.git
git add .
git commit -m "react-create-app on Heroku"
git push heroku master
heroku open
```

That's all there is to it. If you run into problems, check these resources for troubleshooting options:

- Git and GitHub Essentials[228]
- Deploying React with Zero Configuration[229]
- Heroku Buildpack for create-react-app[230]

[226] https://devcenter.heroku.com/articles/heroku-cli
[227] https://www.heroku.com/
[228] https://www.robinwieruch.de/git-essential-commands/
[229] https://blog.heroku.com/deploying-react-with-zero-configuration
[230] https://github.com/mars/create-react-app-buildpack

Outline

So now we've reached the end of The Road to Learn React. I hope you enjoyed reading it, and I hope it helped you to gain some traction in React. If you liked the book, share it as a way to learn React with your friends. It should be used as giveaway. Also, a review on Amazon[231] or Goodreads[232] can really help improve future projects.

So, where can you go from here? I recommend you extend the application on your own, so eventually you can start creating your very own React projects. Consider doing this before you dive into another book, course or tutorial. Do it for one week, take it to production by deploying it, and reach out to me me[233] or others to showcase it. I am always interested in seeing what my readers built and helping them along.

If you are looking for more extensions for your application, I recommend several learning paths after you've mastered the basics:

- **Connecting to a Database and/or Authentication**: In a growing React application, you may want to persist data eventually. The data should be stored in a database so it can survive after a browser session, and so it can be shared across different users using your application. The simplest way to introduce a database is Firebase. In this tutorial[234], you will find a step-by-step guide on how to use Firebase authentication in React. Beyond that, you can use Firebase's realtime database to store user entities.
- **State Management**: You have used React `this.setState()` and `this.state` to manage and access local component state. That's a good start. In a larger application, however, you will experience the limits of React's local component state[235]. It is imperative you learn to use third-party state management libraries like Redux or MobX[236]. On my course platform[237], you will find the course "Taming the State in React" that teaches about advanced local state management using Redux. The course comes with an ebook as well.
- **React Hooks**: You have learned about two kinds of React components: Class and Function Components. In the book it says that Function Components cannot have state management and thus they are called Functional Stateless Components. That's not true anymore, because a recent feature called Hooks in React enables us to use state and side-effects in Function Components. That's why you should learn more about React Hooks[238].

[231] https://amzn.to/2JHlP42
[232] https://www.goodreads.com/book/show/37503118-the-road-to-learn-react
[233] https://twitter.com/rwieruch
[234] https://www.robinwieruch.de/complete-firebase-authentication-react-tutorial/
[235] https://www.robinwieruch.de/learn-react-before-using-redux/
[236] https://www.robinwieruch.de/redux-mobx-confusion/
[237] https://roadtoreact.com/
[238] https://www.robinwieruch.de/react-hooks/

- **Tooling with Webpack and Babel**: We used *create-react-app* to set up the application we created for this book. At some point you may want to learn the tooling around it, which enables you to setup your own project without *create-react-app*. I recommend a minimal setup with Webpack and Babel[239], after which you can apply additional tooling on your own. For instance, you could use ESLint[240] to follow a unified code style.
- **Code Organization**: Recall if you will the chapter about code organization. You can apply these changes now, if you haven't already. It will help organize your components in structured files and folders (modules), and it will help you understand the principles of code splitting, reusability, maintainability, and module API design. Your applications will eventually grow and need to be structured into modules, so it's better to start now.
- **Testing**: The book only scratched the surface of testing. If you are not familiar with the topic, you should dive deeper into unit testing and integration testing, especially with React applications. I would recommend to stick to Enzyme and Jest for implementations, to refine your approaches with unit and snapshot tests.
- **React Component Syntax**: The best practices for implementing React components evolve over time. You will find many ways to write your React components, especially React class components, in other learning resources. A GitHub repository called react-alternative-class-component-syntax[241] is a great way to learn alternative ways to write React class components. By using class field declarations, you can write them even more concisely in the future.
- **UI Components**: Many beginners make the mistake of introducing UI component libraries too early in their projects. It is more practical to learn how to implement and use a dropdown, checkbox, or dialog in React with standard HTML elements. Most of these components will manage their own local state. A checkbox has to know whether it is checked or unchecked, so you should implement them as controlled components. After we covered the foundational implementations, introducing a UI component library with checkboxes and dialogs as React components should be easier.
- **Routing**: You can implement routing for your application with react-router[242]. So far, there is only one page in your application. React Router helps manage multiple pages across multiple URLs. When you introduce routing to your application, you don't make any requests to your web server to request the next page. The router will do everything for you on the client-side. So get your hands dirty and introduce routing in your application.
- **Type Checking**: Earlier, we used React PropTypes to define component interfaces. It is a good practice to prevent bugs, but the PropTypes are only checked on runtime. You can go further by introducing static type checking at compile time. TypeScript[243] and Flow[244] are popular type systems for React.
- **React Native**: React Native[245] brings your application to mobile devices, such as iOS and Android applications. Once you've mastered React, the learning curve for React Native

[239] https://www.robinwieruch.de/minimal-react-webpack-babel-setup/
[240] https://www.robinwieruch.de/react-eslint-webpack-babel/
[241] https://github.com/the-road-to-learn-react/react-alternative-class-component-syntax
[242] https://github.com/ReactTraining/react-router
[243] https://www.typescriptlang.org/
[244] https://flowtype.org/
[245] https://facebook.github.io/react-native/

shouldn't be that steep, as they share the same principles. The only difference with mobile are the layout components.

- **Other Projects**: There are plenty of tutorials out there that use only React to build exciting applications, which provide good practice to apply what you've learned in this book before you move on to intermediate projects. Here are a few of my own:
  - A paginated and infinite scrolling list[246]
  - Showcasing tweets on a Twitter wall[247]
  - Connecting your React application to Stripe for charging money[248].

I invite you to visit my website[249] to find more interesting topics about web development and software engineering. You can also subscribe to my Newsletter[250] to get updates about articles, books, and courses. Furthermore, my course platform[251] offers more advanced lessons about the React ecosystem.

Finally, I hope to find more Patrons[252] who are willing to support my content, as there are many students who cannot afford to pay for educational content. That's why I put lots of my content out there for free. Supporting me on Patreon allows to educate others for free.

Once again, if you liked the book, I ask that you take a moment to think about a person who would like to learn React and share it with them. The book is intended to be given to others, and it will improve over time if more people read it and share feedback.

Thank you for reading the Road to learn React.

Regards,

Robin Wieruch

[246] https://www.robinwieruch.de/react-paginated-list/
[247] https://www.robinwieruch.de/react-svg-patterns/
[248] https://www.robinwieruch.de/react-express-stripe-payment/
[249] https://www.robinwieruch.de
[250] https://www.getrevue.co/profile/rwieruch
[251] https://roadtoreact.com
[252] https://www.patreon.com/rwieruch