by robin wieruch

the Road to learn React

The Road to learn React

Robin Wieruch

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

This version was published on 2017-02-12

Tweet This Book!

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

The suggested tweet for this book is:

I just bought The Road to learn React by @rwieruch #ReactJs #JavaScript

The suggested hashtag for this book is #ReactJs #JavaScript.

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

https://twitter.com/search?q=#ReactJs #JavaScript

Contents

Foreword

I love to teach, even though I am no expert. I learn every day and I have the fortune to have great mentors. After all not everyone has the opportunity to learn from mentors and peers. The book is my attempt to give something back which might help people to get started and advance in React.

But why me? In the past I have written a large tutorial to implement a SoundCloud Client in React + Redux[1]. I never expected the overwhelming reaction. I learned a lot during the process of writing. But even more by getting your feedback. It was my first attempt to teach people in programming.

It also taught me to do better. I realized the SoundCloud tutorial is suited for advanced developers. It uses several tools to bootstrap your application and dives quickly into Redux. Still it helped a lot of people to get started. In my opinion it is a great tutorial to get a bigger picture of React + Redux. I use every free minute to improve the material, but it is time consuming.

In the Road to learn React I want to offer a foundation before you start to dive into the broader React ecosystem. It has less tooling and less external state management, but more React. It explains general concepts, patterns and best practices. You will learn to build your own Hacker News application. It cover real world features like pagination, client-side caching and interactions. Additionally you will transition from JavaScript ES5 to JavaScript ES6. After all the book should give you a solid foundation in React before you dive into more advanced topics like Redux.

[1] https://www.robinwieruch.de/the-soundcloud-client-in-react-redux

FAQ

How do I get updates? You can subscribe[2] or follow me on Twitter[3] to get updates. It keeps me motivated to do my work. Once you have a copy of the book, it will stay updated. But you have to download the copy again once its updated. I will notify everyone via the Newsletter[4] and Twitter[5].

Does it cover Redux? It doesn't. The book should give you a solid foundation before you dive into advanced topics like Redux. Still the implementation in the book will show you that you don't need Redux to build an application.

Can I help to improve it? Yes! You can have a direct impact with your thoughts and contribution on GitHub[6]. I don't claim to be an expert nor to write in native english. I would appreciate your help very much.

What are the reading formats? In addition to the .pdf, .epub, and .mobi formats, you can read it in pure markdown on GitHub[7]. In general, I recommend reading it on a larger format, otherwise the code snippets will have ugly line breaks.

Will you add more chapters in the future? See the Changelog chapter for updates that already happened. In general it depends on the community. If there is an acceptance for the book, I will deliver more chapters and improve old material. I will keep the content up to date with recent best practices, concepts and patterns. But it has to pay me a bit of my effort. I would love to hear your thoughts about possible chapters to improve and enrich the learning experience.

[2] http://eepurl.com/caLPjr
[3] https://twitter.com/rwieruch
[4] http://eepurl.com/caLPjr
[5] https://twitter.com/rwieruch
[6] https://github.com/rwieruch/the-road-to-learn-react
[7] https://github.com/rwieruch/the-road-to-learn-react

Changelog

10. January 2017:

- v2 Pull Request[8]
- even more beginner friendly
- 37% more content
- 30% improved content
- 13 improved and new chapters
- 140 pages of learning material

[8] https://github.com/rwieruch/the-road-to-learn-react/pull/18

What you can expect (so far...)

- Hacker News App in React[9]
- no complicated configurations
- create-react-app to bootstrap your application
- efficient lightweight code
- only React setState as state management (so far...)
- transition from JavaScript ES5 to ES6 along the way
- the React API with setState and lifecycle methods
- interaction with a real world API (Hacker News)
- advanced user interactions
  - client-sided sorting
  - client-sided filtering
  - server-sided searching
- implementation of client-side caching
- higher order functions and higher order components
- snapshot test components with Jest
- unit test components with Enzyme
- neat libraries along the way
- exercises and more readings along the way
- internalize and reinforce your learnings
- deploy your app to production

[9] https://intense-refuge-78753.herokuapp.com/

What you could expect (in the future...)

- advanced components and interactions to build a powerful dashboard
- give your app a structure in terms of files/folders
- arrive at the point to experience how state management could help you
- introduce a state management library to your app
- use common patterns in React and state management
- get to know open source style guides for a better code style
- more neat libraries along the way
- usage of React dev tools and performance profiling
- get to know a diverse set of styling tools in React
- animate your components

How to read it?

Are you new to React? That's perfect. I will need your feedback to improve the material to enable everyone to learn React. You can have a direct impact on GitHub[10] or give me feedback on Twitter[11].

In general each chapter will build up on the previous. Each of them will dive into a new learning. Don't rush through the book. You should internalize each step. You could apply your own implementations and read more about the topic. Make yourself comfortable with the learnings before you continue.

After you have read the book you could dive into the SoundCloud Client in React + Redux[12]. It guides you to implement your own SoundCloud application with a state management library.

Additionally you may want to check out my Tips to learn React (+ Redux)[13].

[10] https://github.com/rwieruch/the-road-to-learn-react

[11] https://twitter.com/rwieruch

[12] https://www.robinwieruch.de/the-soundcloud-client-in-react-redux

[13] https://www.robinwieruch.de/tips-to-learn-react-redux/

Bootstrap your React App

The chapter gives you an introduction to React. You may ask yourself: Why should I learn React in the first place? You will get your answer to that question. Afterwards you will dive into the ecosystem by bootstrapping your first React app. Along the way you will get an introduction to JSX and ReactDOM. Be prepared for your first React components!

Hi, my name is React.

Why should you bother to learn React? In the recent years single page applications (SPA) got popular. Frameworks like Angular, Ember and Backbone helped JavaScript people to build modern web applications beyond jQuery. The list is not exhaustive. There exists a wide range of SPA frameworks. When you consider the release dates, most of them are among the first generation of SPAs: Angular 2010, Backbone 2010, Ember 2011.

The initial React release was 2013 by Facebook. React is no SPA framework but a view library. It only enables you to render components as view in a browser. But the whole ecosystem around React makes it possible to build single page applications.

But why should you consider to use React over the first generation of SPA frameworks? While the first generation of SPAs tried to solve a lot of things at once, React only helps to build your view layer. It's a library and not a framework. The idea behind it: Your view is a hierarchy of composable components.

In React you can focus on your view before you introduce more aspects to your application. Every other aspect is another building block for your SPA. These building blocks are essential to build a mature application. They come with two advantages.

First you can learn the building blocks step by step. You don't have to worry to understand them altogether. It is different to a framework that gives you every building block from the start. The book focuses on React as first building block. More building blocks follow eventually.

Second all building blocks are interchangeable. It makes the ecosystem around React such an innovative place. Multiple solutions are competing with each other. You can pick the most appealing solution for you and your use case.

The first generation of SPA frameworks arrived at an enterprise level. React stays innovative and gets adopted by multiple tech thought leader companies like Airbnb, Netflix and of course Facebook[14]. All of them invest in the future of React and are content with React and the ecosystem itself.

React is probably one of the best choices for building SPAs nowadays. It only delivers the view layer, but the ecosystem is a whole flexible and interchangeable framework. React has a slim API, an amazing ecosystem and a great community. You can read about my experiences why I moved from Angular to React[15]. Everyone is keen to experience where it will lead us in 2017.

[14] https://github.com/facebook/react/wiki/Sites-Using-React

[15] https://www.robinwieruch.de/reasons-why-i-moved-from-angular-to-react/

Requirements

Before you start to read the book, you should be familiar with HTML, CSS and JavaScript (ES5). The book will teach ES6 and beyond. When you come from a different SPA framework or library, you should be familiar with the basics. When you just started in web development, you should feel comfortable with HTML, CSS and JavaScript ES5 to learn React.

Every developer needs tools to build applications. You will need an editor and terminal (command line) tool. You can read my developer setup to organize your tools: Developer Setup[16]. The editor is used to organize and write your code. The terminal is used to execute commands. A command can be to start your application, to run tests or to install other libraries for your project.

Last but not least you will need an installation of node and npm[17]. Both are used to manage libraries you will need along the way. 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 or npm first. These are my versions:

```
node --version
*v5.0.0
npm --version
*v3.3.6
```

[16] https://www.robinwieruch.de/developer-setup/
[17] https://nodejs.org/en/

node and npm

The chapter gives you a little crash course in node and npm. If you are familiar with both of them, you can skip the lesson.

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

Global packages are accessible from everywhere in the terminal and you have to install them only once. You can install a global package easily by typing:

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

Local packages are used in your application. For instance React will be a local package which can be required in your application for usage. You can install it via the terminal by typing:

```
npm install <package>
```

In the case of React it would be:

```
npm install react
```

The installed package folder will automatically appear in a folder called *node_modules/*.

But be careful! Whenever you install a local package you shouldn't forget the neat `--save` configuration:

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

The `--save` indicates npm to store the package requirement to a file called *package.json*. The file can be found in your project folder.

Not every project folder comes with a *package.json* though. There is a npm command to initialize a *package.json*. Only when you have that file, you can install new local packages via npm.

```
npm init -y
```

The `-y` is a shortcut to initialize all the defaults in your *package.json*. If you do it without it, you have to interactively decide how to configure the file.

One more word about the *package.json*. The file enables you to share your project with other developers without sharing all the packages. The file has all the references of node packages used in your project. These packages are called dependencies. Everyone can copy your project without the dependencies. The dependencies are references in the *package.json*. Someone who copies your project can install all packages via `npm install` on the command line.

I want to cover one more npm command to prevent confusion:

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

The `--save-dev` indicates that the node package is only used in the dev environment. It will not be used in production. What kind of node package could that be? Imagine you want to test your application with an external library. You need to install that library via npm, but want to exclude it from your production environment. There you don't want to test your application anymore. It should be tested already and work out of the box for users.

You will encounter more npm commands on your way. But these will be sufficient for now.

Exercises:

- read more about npm[18]

[18] https://docs.npmjs.com/

Installation

There are multiple approaches to get started with a React application.

The first one is to use a CDN. That may sound more complicated than it is. A CDN is a content distribution network. Several companies have CDNs which host files publicly for users. These files can be whole libraries like React.

How to use it? You can inline the `<script>` in your HTML that points to a CDN address. To get started in React you need two files (libraries): react and react-dom.

```
<script src="https://unpkg.com/react@15/dist/react.js"></script>
<script src="https://unpkg.com/react-dom@15/dist/react-dom.js"></script>
```

But why should you use a CDN when you have npm to install packages? When your application has a *package.json* file, you can install react and react-dom from the command line. You can install multiple packages in one line with npm.

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

That approach is mostly used to add React to an existing application.

Unfortunately that's not everything. You would have to deal with Babel[19] to make your application aware of JSX (React syntax) and JavaScript ES6. Babel transpiles your code that Browsers can interpret ES6 and JSX. Not all browser are capable to interpret the novel syntax. The setup includes a lot of configuration and tools. It can be overwhelming for React beginners to bother with all the configuration. That's why Facebook introduced create-react-app!

Exercises:

- read more about React installation[20]

[19] http://babeljs.io/

[20] https://facebook.github.io/react/docs/installation.html

create-react-app

You will use create-react-app[21] to bootstrap your app. It's an opinionated yet zero-configuration starter kit for React introduced by Facebook. People like and would recommend it to beginners by 96%[22]. In create-react-app the tools and configuration evolve in the background while the focus is on the application implementation.

To get started you will have to install the package to your global packages. After that you always have it available on the command line to bootstrap new React applications.

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

You can check the version of create-react-app to verify the installation in your command line:

```
create-react-app --version
```

Now you can bootstrap your first React application. We call it *hackernews*, but you can choose a different name. Afterwards simply navigate into the folder:

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

Now you can open the app in your editor. The following folder structure will be presented to you:

```
hackernews/
  README.md
  node_modules/
  package.json
  .gitignore
  public/
    favicon.ico
    index.html
  src/
    App.css
    App.js
    App.test.js
    index.css
    index.js
    logo.svg
```

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

[22] https://twitter.com/dan_abramov/status/806985854099062785

In the beginning everything you need is located in the *src* folder.

The main focus lies on the *App.js* file to implement React components. It will be used alone, but later you might want to split up your components into multiple files.

Additionally you will find a *App.test.js* file for tests and a *index.js* as entry point to the React world. You will get to know both files in a later chapter. Moreover there is a *App.css* and *index.css* file to style your application. They all come with default implementations when you open them.

Next to to the *src* folder you will find the *package.json* file and *node_modules/* folder to manage your node packages.

Additionally create-react-app comes with the following npm scripts for your command line:

```
// Runs the app in http://localhost:3000
npm start

// Runs the tests
npm test

// Builds the app for production
npm run build
```

The scripts are defined in your *package.json* too. You can read more about the scripts and create-react-app[23] in general.

Exercises:

- `npm start` your app and visit the page in your browser
- run the interactive `npm test` script
- make yourself familiar with the folder structure
- make yourself familiar with the content of the files

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

Introduction to JSX

Now you will get to know JSX - the syntax in React. As mentioned create-react-app already scaffolded a boilerplate application. All files come with default implementations. Let's dive into the source code. The only file you will touch in the beginning will be the *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">
        <div className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <h2>Welcome to React</h2>
        </div>
        <p className="App-intro">
          To get started, edit <code>src/App.js</code> and save to reload.
        </p>
      </div>
    );
  }
}

export default App;
```

Don't let yourself confuse by the import and export statements. We will revisit those later.

In the file you have an **ES6 class component** with the name App. Basically you can use the <App /> JSX everywhere in your application to use the component. Once you use it, it will produce an **instance** of your **component**. The **element** it returns is specified in the `render()` method. Elements are what components are made of. It is valuable to understand the differences between component, instance and element.

Pretty soon you will see where the App component is used. Otherwise you wouldn't see the rendered output in the browser, would you? The App component is only the declaration, but not the usage. You would use the component somewhere else with `<App />`.

The content in the render block looks pretty similar to HTML, but it's JSX. JSX allows you to mix HTML and JavaScript. It's powerful yet confusing when you are used to plain HTML. That's why a good starting point is to use basic HTML in your JSX. Next you can start to embed JavaScript expressions in between by using curly braces.

First let's remove all the distracting clutter.

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

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

export default App;
```

Now you only return HTML without JavaScript. Let's make the a "Welcome to React" variable that will be used in your JSX.

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

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

export default App;
```

It should work when you start your application on the command line.

Additionally you might have noticed the `className` attribute. It reflects the `class` HTML attribute. Because of technical reasons JSX had to replace some internal HTML attributes. You can find all of the supported HTML attributes in the React documentation[24].

[24] https://facebook.github.io/react/docs/dom-elements.html

Exercises:

- read more about JSX[25]
- read more about React components, elements and instances[26]
- define more variables to render them in your JSX

[25] https://facebook.github.io/react/docs/introducing-jsx.html
[26] https://facebook.github.io/react/blog/2015/12/18/react-components-elements-and-instances.html

ES6 const and let

I guess you noticed that we declared the variable `helloWorld` with `var`. JavaScript ES6 comes with two more options to declare your variables: const and let. In ES6 JavaScript you will rarely find var anymore. Let's get some explanation for const and let.

A variable declared with `const` cannot be re-assigned or redeclared. It can't get mutated (changed, modified). You embrace immutable data structures by using it.

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

A variable declared with `let` can get mutated.

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

However you have to be careful at one point with const. A variable declared with const cannot get modified. However when the variable is an array or object the value it holds can get altered. The value is not immutable.

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

But when to use each declaration? There are different opinions about the usage. I suggest to use const whenever you can. It indicates that you want to keep your data structure immutable even though values in objects and arrays can get altered. Still immutability is embraced in React and its ecosystem. When you want to modify your variable, you should use let over const.

Let's use const over var in the App component.

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

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

export default App;
```

Exercises:

- read more about ES6 const[27]
- read more about ES6 let[28]

[27] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const
[28] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let

ReactDOM.render

Before you continue with the App component, you might want to see where it's used. It is located in your entry point to the React world *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')
);
```

Basically `ReactDOM.render()` uses a DOM node in your HTML to replace it with your JSX. That's how you can easily integrate React in every foreign app. It is not forbidden to use `ReactDOM.render()` multiple times. You can use it at multiple places in your application to bootstrap simple JSX syntax, a React component, multiple React components or a whole application.

ReactDOM.render expects two arguments.

The first argument is JSX that gets rendered. The second argument specifies the place where the React application hooks into your HTML. It expects an element with an `id='root'`. Open your *public/index.html* file to find the id.

In the implementation `ReactDOM.render()` already takes your App component. However it would be fine to pass simpler JSX as long as it is JSX.

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

Exercises:

- read more about React rendering element[29]

[29] https://facebook.github.io/react/docs/rendering-elements.html

Complex JavaScript in JSX

Let's get back to your App component. So far you rendered some primitive variables in your JSX. Now you will start to render a list of items. The list will be artificial data in the beginning, but later you will fetch the data from an external API. That will be far more exciting.

First you define the list of items.

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

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

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

Second you can use the built-in map functionality in your JSX. It enables you to iterate over your list of items to display them. As mentioned you will use curly braces to encapsulate the JavaScript expression in your JSX.

```
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;
```

You can see how the map functionality is simply inlined. Each item property is display in a `<span>` tag. Moreover the url property of the item is used in the `href` attribute of the anchor tag.

React will do all the work for you and display each item. But you should add one helper for React to embrace its full potential. You have to assign a key attribute to each list element. Only that way React is able to identify added, changed and removed items when the list changes.

```
{ 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 the key attribute a stable identifier. Don't make the mistake of using the array index, which isn't stable. React will have a hard time identifying the items properly when their order changes.

```
// bad example
{ list.map(function(item, key) {
  return (
    <div key={key}>
      ...
    </div>
  );
})}
```

Start your app, open your browser and see both items of the list displayed.

Exercises:

- read more about React lists and keys[30]
- make yourself comfortable with standard built-in functionalities in JavaScript[31]
- use more JavaScript expression on your own in JSX

[30] https://facebook.github.io/react/docs/lists-and-keys.html
[31] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map

ES6 Arrow Functions

JavaScript ES6 introduced arrow functions. An arrow function expression is shorter than a function expression.

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

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

But you have to be aware of its functionalities. One of them is a different behavior with the `this` object. A function expression always defines its own `this` object. Arrow function expressions still have the `this` object of the enclosing context. Don't get confused when using `this` in an arrow function.

There is another valuable fact about arrow functions regarding the parenthesis. You can remove the parenthesis when the function gets only one argument, but have to keep them when it gets multiple arguments.

```
item => { ... }

(item) => { ... }

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

However let's have a look at the map function. You can write it more concise with an ES6 arrow function.

```
{ 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>
  );
})}
```

Additionally you can remove the block body. In a concise body an implicit return is attached thus you can remove the return statement.

```
{ 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 looks more readable again. It omits the function and return statements.

Exercises:

- read more about ES6 arrow functions[32]

[32] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/Arrow_functions

Your *src/App.js* should look like the following by now:

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

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

class App extends Component {
  render() {
    return (
      <div className="App">
        { 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>
    );
  }
}
```

```
export default App;
```

You have learned to bootstrap your own React app! Let's recap the last chapters:

- React
  - create-react-app bootstraps a React app
  - JSX mixes up HTML and JavaScript to express React components
  - difference between components, instances and elements
  - ReactDOM.render() is an entry point for a React app
  - built-in JavaScript functionalities like map can be used to render a list of items
- ES6
  - variable declarations with const and let
  - arrow functions to shorten your function declarations

It makes sense to make a break at this point. Internalize the learnings and apply them on your own. You can experiment with the source code you have written so far.

Basics in React

The chapter will guide you through the basics of React. It covers state and interactions in components - because static components are a bit dull, aren't they? Additionally you will learn about the options to declare a component and how to keep components composeable and reusable. Be prepared to breath life into your components.

Internal Component State

Internal component state allows you to store, modify and delete properties of your component. The ES6 class component can use a constructor to initialize internal component state. The constructor is called only once when the component initializes. Don't worry about the constructor and ES6 class itself - we will cover these in later chapters.

Let's introduce a class constructor where you can set the initial internal component state.

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      list: list,
    };

  }

  ...

}
```

In your case the initial state is the list of items. Note that you have to call `super(props);` to call the constructor of the extended class. It's mandatory.

The state is bound to the class with the `this` object. You can access the state in your component - for instance in the `render()` method. Before you have mapped a static list of items. Now you are about to use the list of your internal component state.

```
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. It resides in the internal component state. You could add items, change items or remove items in and from your list. Every time you change your component state, the `render()` method of your component will run again. But be careful! Don't mutate the state directly, you have to use a method called `setState()` to modify your state. You will get to know it in a following chapter.

Exercises:

- read more about the ES6 class constructor[33]

[33] 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 concise. Imagine the following object initialization:

```
const name = 'Robin';

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

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

```
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.

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

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

Another neat helper are shorthand method names. In ES6 you can initialize methods in an object more concise.

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

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

Last but not least you are allowed to use computed property names in ES6.

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

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

Computed property names might make no sense yet. Why should you need a computed property name? But you will come to a point where you can use it.

Exercises:

- read more about ES6 object initializer[34]

[34] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Object_initializer

Unidirectional Data Flow

Now you have some internal state in your App component. However you have not manipulated the internal state yet. The state is static and thus is the component. A good way to experience state manipulation is to have some component interaction.

Let's add a button in our displayed list of items. The button says "Dismiss" and will remove the item from the list. It could be useful onetime when you only want to keep a list of unread items.

```
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>
    );
  }
}
```

As you can see the `onDismiss()` method gets enclosed by another function. Only that way you can sneak in the `objectID`. Otherwise you would have to define the function outside of the elements. However by using an arrow function you can inline it.

Note that elements with multiple attributes get messy as one line at some point. That's why the button is already used with multilines and indentation to keep it readable. But it is not mandatory. It is only a code style recommendation.

Now you have to implement the `onDismiss()` functionality. It takes an item id to identify the item to dismiss. The function is bound to the class and thus becomes a class method. You have to bind class methods in the constructor. Additionally you have to define its functionality in your class.

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      list,
    };

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

  onDismiss(id) {
    ...
  }

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

Now you can define what happens inside of the class method. Since you want to remove the clicked item from your list, you can do that with the built-in array filter functionality. The filter function takes a function to evaluate each item in the list. If the evaluation for an item is true, the item stays in the list. Otherwise it will be removed. Additionally the function returns a new list and doesn't mutate the old list.

```
onDismiss(id) {

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

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

You can do it more concise by using an arrow function again.

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

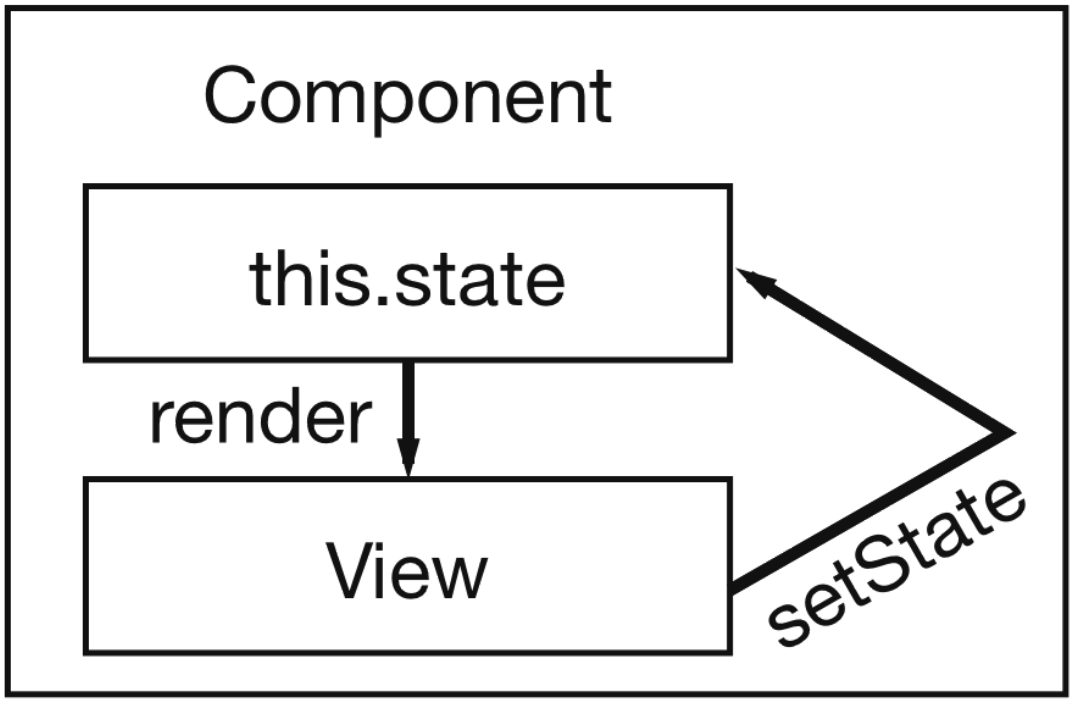
You could even inline it - like we did in the `onClick()` handler of the button - but it might get less readable.

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

The list removes the clicked item now. However the state isn't updated yet. You can use the `setState()` class method to update the list in the internal component state.

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

Now run again your app and try the "Dismiss" button. It should work. What you experience now is the **unidirectional data flow** in React. You trigger an action in your view - with `onClick()` - a function or class method modifies the internal component state and the `render()` method of the component runs again to update the view.



Internal state update with unidirectional data flow

Exercises:

- read more about the state and lifecycle in React[35]

[35] https://facebook.github.io/react/docs/state-and-lifecycle.html

Interactions with Forms and Events

Let's add another interaction to experience forms and events in React. The interaction is a search functionality. The input of the search field should be used to filter your list based on the title property of an item.

First you define your input field in your JSX.

```
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 field and filter the list temporary by the search term. To be able to filter the list, you need the value of the input field to update the state. But how do you access the value? You can use synthetic events in React to access the event payload.

Let's define an `onChange()` function for the input field.

```
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 and thus a class method again. You have to bind and define the method.

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      list,
    };

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

  onSearchChange() {
    ...
  }

  ...
}
```

The method gives you access to the synthetic React event. The event has the value of the input field in its target object. Now you can manipulate the state for the search term:

```
class App extends Component {

  ...

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

  ...
}
```

Additionally you have to define the initial state for the `searchTerm` in the constructor.

```
class App extends Component {

  constructor(props) {
    super(props);

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

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

  ...
}
```

Now you store the input value to your internal component state every time the value in the input field changes. However the list doesn't update yet. You have to filter the list temporary based on the `searchTerm`. That's fairly simple. Before you map the list you can apply a filter on it. You have already used the built-in filter functionality.

```
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 now. We want to define the filter argument - the function - outside of our ES6 class component. There we don't have access to the state of the component - thus we have no access to the `searchTerm` property to evaluate the filter condition. We have to pass the `searchTerm` to the filter function and have to return a new function to evaluate the condition. That's called a higher order function.

Normally I wouldn't mention higher order functions, but in a React book it makes totally sense. It makes sense to know about higher order functions, because React deals with a pattern called higher order components. You will get to know the pattern later. Now again let's focus on the filter functionality.

First you have to define the higher order function outside of your class.

```
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 which takes an item. The returned function will be used to filter the list based on the condition defined in the function.

Let's define the condition.

```
function isSearched(searchTerm) {
  return function(item) {
    return !searchTerm || item.title.toLowerCase().includes(searchTerm.toLowerCa\
se());
  }
}

class App extends Component {

  ...

}
```

The condition says multiple things. You filter the list only when a `searchTerm` is set. When a `searchTerm` is set, you match the incoming `searchTerm` pattern with the title of the item. You can do that with the built-in functionality includes. Only when the pattern matches you return true and the item stays in the list. But be careful with pattern matching: You shouldn't forget to lower case both strings. Otherwise there will be mismatches between a search term 'redux' and an item title 'Redux'.

One thing is left to mention: We cheated a bit by using the built-in includes functionality. It is already an ES6 feature. How would that look like in ES5? You would use the `indexOf()` function to get the index of the item in the list. When the item is in the list, `indexOf()` will return a positive index.

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

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

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

```
// ES5
function isSearched(searchTerm) {
  return function(item) {
    return !searchTerm || item.title.toLowerCase().includes(searchTerm.toLowerCa\
se());
  }
}

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

One could argue which one is more readable. Personally I prefer the second one. The React ecosystem uses a lot of functional programming concepts. It happens often that you will use a function which returns a function (higher order functions). In ES6 you can express these more concise with arrow functions.

Last but not least you have to use the defined `isSearched()` function to filter your list.

```
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.

Exercises:

- read more about React events[36]
- read more about higher order functions[37]

[36] https://facebook.github.io/react/docs/handling-events.html
[37] https://en.wikipedia.org/wiki/Higher-order_function

ES6 Destructuring

There is a way in ES6 to access properties in objects and arrays easier. It's called destructuring. Compare the following snippet in ES5 and ES6.

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

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

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

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

While you have to add an extra line each time you want to access an object property in ES5, you can do it in one line in ES6. Additionally you don't have to have duplicated property names. A best practice for readability is to use multilines when you destructure an object into multiple properties.

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

The same goes for arrays. You can destructure them too, but keep it more readable with multilines.

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

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

Perhaps you have noticed that the state in the App component can be destructure the same way. You can keep the list filter and map line of code more concise.

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

Again you could do it the ES5 or ES6 way:

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

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

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

Exercises:

- read more about ES6 destructuring[38]

[38] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

Controlled Components

You already learned about the unidirectional data flow. The same law applies for the input field, which updates the state that in turn filters the list. The state was changed, the `render()` method runs again and uses the recent `searchTerm` state to apply the filter condition.

But didn't we forget something in the input element? A HTML input tag comes with a `value` attribute. The value attribute usually has the value that is shown in the input field - in our case the `searchTerm` property. However it seems like we don't need that in React.

That's wrong. Form elements such as <input>, <textarea> and <select> hold their own state. 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. In React you should make sure to make those elements **controlled components**.

How to do that? You only have to set the value attribute of the input field. The value is already saved in the `searchTerm` state property.

```
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 cycle for the input field is self-contained now. The internal component state is the single source of truth for the input field.

The whole internal state management and unidirectional data flow might be new to you. But once you are used to it, it will be your natural flow to implement things in React. In general React brought a novel pattern with the unidirectional data flow. It gets adopted by several frameworks and libraries.

Exercises:

- read more about React forms[39]

[39] https://facebook.github.io/react/docs/forms.html

Breakup Your Components

You have one large App component. It keeps growing and gets confusing eventually. You can start to split it up into chunks - smaller components.

Let's start to use a component for the search input and a component for the list of items.

```
class App extends Component {

  ...

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

You can pass those components properties which they can use themselves.

```
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 you can define the components next to your App component. Those components will be ES6 class components as well. They render the same elements like before.

The first one is the Search component.

```
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.

```
...

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. Perhaps you have seen the `this.props` object. The props - short form for properties - have all the values you have passed to the components when you used them in your App component. You could reuse these components somewhere else but pass them different values. They are reusable.

Exercises:

- figure out which components you could split up
  - but don't do it, otherwise you will run into conflicts in the next chapters

Composeable Components

There is one more little property which is accessible in the props object - the `children` prop. You can use it to pass elements to your components from above - which are unknown to the component itself - but make it possible to compose components into each other. Let's see how this looks like when you only pass text as child to the Search component.

```
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 props. Then it can specify where the children should be displayed.

```
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 be visible next to your input field now. When you use the Search component somewhere else, you can choose a different text if you like. After all it is not only text that you can pass as children. You can pass an element and element trees (which can be encapsulated by components again) as children. The children property makes it possible to weave components into each other.

Exercises:

- read more about the composition model of React[40]

[40] https://facebook.github.io/react/docs/composition-vs-inheritance.html

Reusable Components

Reusable and composeable components empower you to come up with capable component hierarchies. They are the foundation of your view layer. The last chapters mentioned often the term reusability. You can reuse the Table and Search components already. Not to forget the App component.

Let's define one more reusable component - a Button component - which gets reused more often eventually.

```
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 such a component. You will use a `Button` instead of a `button`. It only spares the `type="button"`. Except for the type attribute you have to define everything else when you want to use the Button component. But you have to think about the long term investment here. Imagine you have several buttons in your app, but want to change an attribute, style or behavior for the button. Without the component you would have to refactor every button. Instead the Button component ensures to have only one single source of truth. One Button to refactor all buttons at once.

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

```
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. But we didn't pass any className when the Button was used. It should be more explicit in the Button component that the className is optional. Thus you can use an ES6 default parameter.

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

    ...
  }
}
```

Now whenever there is no `className` property the value will be an empty string.

Exercises:

- read more about ES6 default parameters[41]

[41] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/Default_parameters

Component Declarations

By now you have four ES6 class components. But you can do better. Let me introduce functional stateless components as replacements for ES6 class components. Before you will refactor your components, let's introduce the three different types of components.

- Functional Stateless Components: These components are functions which get an input and return an output. The input is state (properties). The output is a component instance. Same as in an ES6 class component. Functional stateless components are functions (functional) and they have no internal state (stateless). You cannot access the state with `this.state` because there is no `this` object. Additionally they have no lifecycle methods. You didn't learn about lifecycle methods yet, but you already used two: `constructor()` and `render()`. Keep this in mind, when you arrive at the lifecycle methods chapter later on.
- ES6 Class Components: You already used these type of components. In the class definition they extend from a React component. The `extend` hooks all the lifecycle methods - available in the React component API - to the component. As mentioned you already used two of them. Additionally you can store and manipulate state in ES6 class components.
- React.createClass: The component declaration was used in older versions of React and still in ES5 React applications. But Facebook declared it as deprecated[42] in favor of ES6.

But when to use functional stateless components or ES6 class components? A rule of thumb is to use functional stateless components when you don't need internal component state or component lifecycle methods. Usually you start to implement your components as functional stateless components. Once you need access to the state or lifecycle methods, you have to refactor it to an ES6 class component.

Let's get back to our application. The App component uses internal state. That's why it has to stay as an ES6 class component. But the other three of your ES6 class components are stateless without lifecycle methods. Let's refactor the Search component to a stateless functional component. The Table and Button component refactoring will remain as your exercise.

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

[42] https://facebook.github.io/react/blog/2015/03/10/react-v0.13.html

```
    </form>
  );
}
```

You already know about the props destructuring. The best practice is to use it in the function signature in the first place.

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

But it can get better. You know already that arrow functions allow you to keep your functions concise. You can remove the block body of the function. In a concise body an implicit return is attached thus you can remove the return statement. Since your functional stateless component is a function, you can keep it concise as well.

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

The last step was especially useful to enforce only to have props as input and an element as output. Nothing in between. Still you could *do something* in between by using a block body in your arrow function.

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

  // do something

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

But you don't need it for now. That's why you can keep the previous version without the block body.

Now you have one lightweight functional stateless component. Once you would need access to its internal component state or lifecycle methods, you would refactor it to an ES6 class component.

Exercises:

- refactor the Table and Button component to stateless functional components
- read more about ES6 class components and functional stateless components[43]

[43] https://facebook.github.io/react/docs/components-and-props.html

Styling Components

Let's add some basic styling to your app and components. You can reuse the *src/App.css* and *src/index.css* files. I prepared some CSS to copy and paste, but feel free to use your own style.

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;
  outline: none;
  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;
}

*:focus {
  outline: none;
}
```

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 you can use the style in your some of your components. Don't forget to use React `className` instead of `class` as attribute.

First apply it in your App ES6 class component.

```
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.

```
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 you have styled your app and components with basic CSS. It should look decent. As you know JSX is mixed up HTML and JavaScript. One could argue to add CSS in the mixup as well. That's called inline style. 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.

```
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>
```

It is really inlined now. You could define the style objects outside of your elements to make it cleaner.

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

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

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

After that you could use it in your columns: `<span style={smallColumn}>`.

In general you will find different opinions and solutions for style in React. You used pure CSS and Inline Style now. It's sufficient to get started.

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. I'm open to your thoughts about the options as well.

- radium[44]
- aphrodite[45]
- styled-components[46]
- CSS Modules[47]

The chapter might get overhauled in the future to give you an opinionated approach.

[44] https://github.com/FormidableLabs/radium
[45] https://github.com/khan/aphrodite
[46] https://github.com/styled-components/styled-components
[47] https://github.com/css-modules/css-modules

Your *src/App.js* should look like the following by now:

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

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

const isSearched = (searchTerm) => (item) =>
  !searchTerm || item.title.toLowerCase().includes(searchTerm.toLowerCase());

class App extends Component {

  constructor(props) {
    super(props);

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

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

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

```
  }

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

  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>
    );
  }
}


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

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>

const Button = ({ onClick, className = '', children }) =>
  <button
    onClick={onClick}
    className={className}
    type="button"
  >
    {children}
  </button>

export default App;
```

You have learned the basics to write your own React app! Let's recap the last chapters:

- React
  - this.state and setState to manage your internal component state
  - forms and events in React

  - unidirectional data flow
  - compose components with children and reusable components
  - usage and implementation of ES6 class components and functional stateless components
  - approaches to style your components
- ES6
  - arrow functions with block and concise bodies to shorten your function declarations
  - 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 learnings and apply them on your own. You can experiment with the source code you have written so far. Additionally you can read more in the official documentation[48].

[48] https://facebook.github.io/react/docs/installation.html

Getting Real with an API

Now it's time to get real with an API. In the end it can get boring to deal with artificial data. Do you know Hacker News[49]? It's a great news aggregator. You will use the Hacker News API to fetch trending stories from the platform. There is a basic[50] and search[51] API. The latter one makes sense in your case to search stories on Hacker News. You can visit the API specification[52] to get a glimpse of the data structure.

[49] https://news.ycombinator.com/
[50] https://github.com/HackerNews/API
[51] https://hn.algolia.com/api
[52] https://hn.algolia.com/api

Lifecycle Methods

You will need the knowledge about React lifecycle methods before you can start to fetch data. These methods are a hook into the lifecycle of a React component which you can overwrite. There are a few to learn.

You already know two lifecycle methods in your ES6 class component: `constructor()` and `render()`.

The constructor is only called when an instance of the component is created and inserted in the DOM. That process is called mounting of the component.

The `render()` method is called during the mount process too, but also when the component updates. Each time when the state of a component changes the `render()` method is called. It also renders when there are incoming props from the component above.

Now you know two lifecycle methods and when they are called. You already used them as well. But there are more of them.

The mounting of a component has two more lifecycle methods: `componentWillMount()` and `componentDidMount()`. The constructor is called first, `componentWillMount()` gets called before the `render()` method, and `componentDidMount()` is called after the `render()` method.

Overall the mounting process has 4 lifecycle methods. They are invoked in the following order:

- constructor()
- componentWillMount()
- render()
- componentDidMount()

But what about the update lifecycle of a component? Overall it has 5 lifecycle methods in the following order:

- componentWillReceiveProps()
- shouldComponentUpdate()
- componentWillUpdate()
- render()
- componentDidUpdate()

You don't need to know all of them from the beginning. But still it is good to know that each lifecycle method can be used for specific use cases:

- componentDidMount() method can be used to get data from an API when the component mounts

- shouldComponentUpdate() can be used in a mature React app to prevent a component from updating for performance optimizations

So far you know that there is a mounting and updating lifecycle. But every lifecycle ends at some time. The third lifecycle is the unmounting lifecycle. It has only one lifecycle method: `componentWillUnmount()`. It is used to cleanup stuff when you are about to destroy your component.

The `constructor()` and `render()` lifecycle methods are already used by you. These are the commonly used lifecycle methods for ES6 class components. Actually the `render()` method is required - otherwise you wouldn't return a component instance.

Exercises:

- read more about lifecycle methods in React[53]
- read more about the state and lifecycle in React[54]

[53] https://facebook.github.io/react/docs/react-component.html
[54] https://facebook.github.io/react/docs/state-and-lifecycle.html

Fetch Data from an API

Now you are prepared to fetch data from the Hacker News API. I mentioned one lifecycle method that can be used to fetch data: componentDidMount(). Before we can use it, let's set up the url constants and default parameters to breakup the API request into chunks.

```
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 ES6 JavaScript you can use template strings[55] to concatenate strings. You will use it to concatenate your url for the API endpoint.

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

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

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

That will keep your url composition flexible in the future.

But let's get to the API fetch where we use the url. The whole data fetch process will be presented at once, but each step will get explained afterwards.

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

```
...

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

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

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

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

  ...
}
```

A lot of things happened. I thought about it to break it into smaller pieces. Then again it would be difficult to grasp the relations of each piece to each other. Let me explain each step in detail.

First, you can remove the artificial list of items, because you return a result from the Hacker News API. The initial state of your component has an empty result and default search term. The same default search term is used in the search field and in your first request.

Second, you use the `componentDidMount()` lifecycle method to fetch the data after the component

did mount. In the very first fetch the default search term from the component state is used. It will fetch "redux" related stories, because that is the default parameter.

Third, the native fetch is used. ES6 string template strings allow it to compose the url with the `searchTerm`. The url is the argument for the native fetch API function. The response needs to get transformed to Json and can finally be set in the internal component state.

Don't forget to bind your new component methods.

Now you can use the fetched data instead of the artificial list of items. However you have to be careful again. The result is not only a list of data. It's a complex object with meta information and a list of hits (stories).[56] You can output the internal state with `console.log(this.state);` in your `render()` method to visualize it.

Let's use the result and render it. But we will prevent to render anything - return null - when there is no result. Once the request to the API succeeded, the result is saved to the state, the App component will render the component instance.

```
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>
    );
  }
}
```

Let's recap what happens during the component lifecycle. Your component gets initialized by the constructor. After that it renders for the first time. But we prevent to display it, because the result is empty. Then the `componentDidMount()` lifecycle method runs. In that method you fetch the

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

data from the Hacker News API asynchronously. Once the data arrives, it changes your internal component state. After that the update lifecycle comes into play. The component runs the `render()` method again, but this time with populated data in your internal component state. The component and thus the Table gets rendered.

The list of hits should be visible now. But the "Dismiss" button is broken. We will fix that soon.

Exercises:

- read more about ES6 template strings[57]
- read more about the native fetch API[58]
- experiment with the Hacker News API[59]

[57] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Template_literals
[58] https://developer.mozilla.org/en/docs/Web/API/Fetch_API
[59] https://hn.algolia.com/api

ES6 spread operator

The "Dismiss" button doesn't work because the `onDismiss()` method is not aware of the result. Let's change that:

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

But what happens in `setState()` now? Unfortunately the result is a complex object. The list of hits is only one of multiple properties in the object. However only the list gets updated - when an item gets removed - in the result object while the other properties stay the same.

One approach could be to mutate the hits in the result object. I will demonstrate it, but we won't do it that way.

```
this.state.result.hits = updatedHits;
```

React embraces functional programming. Thus you shouldn't mutate an object (or mutate the state directly). A better approach is to generate a new object based on the old object and updated object. Thereby none of the objects get altered. These data structures in React are called immutable data structures. They always return a new object and never alter an object.

Let's do it in JavaScript ES5. `Object.assign()` takes as first argument a target object. All following arguments are source objects. These objects are merged into the target object. The target object can be an empty object - it embraces immutability, because no source objects get mutated. It would look similar to the following:

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

Now let's do it in the `onDismiss()` method:

```
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 it in ES5. There is a simpler solution in ES6 and future JavaScript releases. May I introduce the spread operator to you? It only consists of three dots: ... When it is used, every value from an array or object gets copied to another array or object.

Let's examine the ES6 **array** spread operator even though you don't need it yet.

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

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

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

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

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

Now let's have a look at the object spread operator. It is not ES6! It is a proposal for a future ES version[60] yet already used by the React community. That's why create-react-app incorporated the feature in the configuration.

Basically it is the same as the ES6 array spread operator but with objects. It copies each key value pair into a new object.

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

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

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

Multiple objects can be spread like in the array spread example.

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

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

After all it can be used to replace ES5 `Object.assign()`.

```
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 again.

Exercises:

- read more about Object.assign()[61]
- read more about the ES6 array spread operator[62]
  - the object spread operator is briefly mentioned

[61] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Object/assign
[62] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Spread_operator

Conditional Rendering

The conditional rendering is introduced pretty early in React applications. It happens when you want to make a decision to render either one or another element. Sometimes it means to render an element or nothing. After all a conditional rendering can be expressed by a if-else condition in JSX.

The result in the internal component state is null in the beginning. So far the App component returned no component instance when the result hasn't arrived from the API. That's already a conditional rendering! But it makes more sense to wrap the Table component - which depends solely on the result - in an independent conditional rendering. Everything else should be displayed even though there is no result yet. You can simply use a ternary expression in your JSX.

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

That's only one option to express a conditional rendering. Another option is the logical `&&` operator.

In JavaScript a `true && 'Hello World'` always evaluates to 'Hello World'. A `false && 'Hello World'` always evaluates to false.

In React you can see it similar. If the condition is true, the expression right after && will be the output. If the condition is false, React ignores and skips the expression. It is applicable in the Table conditional rendering case, because it should return a Table or nothing.

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

These were a few approaches to use conditional rendering in React. After all you should be able to see the fetched data in your list. Everything except the Table is displayed when the data fetching is pending.

Exercises:

- read more about React conditional rendering[63]

[63] https://facebook.github.io/react/docs/conditional-rendering.html

Client- or Server-side Interaction: Search

When you use the search input field now, you will filter the list. That's happening on the client-side though. Now you are going to use the Hacker News API to search on the server-side. Otherwise you would deal only with the first API response which you got on `componentDidMount()` with the default search term parameter.

You can define an `onSubmit()` method in your ES6 class component, which fetches results from the Hacker News API. It will be the same fetch like in your `componentDidMount()` lifecycle method. But it fetches it with the modified search term from the search field input.

```
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);
  }

  ...

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

  ...
}
```

The Search component gets extended with a button. The button has to explicitly trigger the search. Otherwise you would fetch data every time from the Hacker News API when your input changes. As alternative you could debounce (delay) the `onChange()` function and spare the button, but it would add more complexity at this time. Let's keep it without debounce.

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

```
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 in the button.

```
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 anymore. The result comes directly from the Hacker News API after you have clicked the Search button.

```
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>
```

When you try to search now, you will experience that the browser reloads. That's a native browser

behavior of submit in a form. In React you will often come across the `preventDefault()` event method to suppress the native browser behavior.

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

Now you should be able to search different Hacker News stories. You interact with a real world API. There should be no client-sided search anymore.

Exercises:

- read more about synthetic events in React[64]

[64] https://facebook.github.io/react/docs/events.html

Paginated Fetch

Did you have a closer look at the returned data structure yet? The Hacker News API[65] returns more than a list of hits. The page property - which is 0 in the first response - can be used to fetch more paginated data. You only need to pass the next page with the same search term to the API.

First let's extend the composeable API constants that it can deal with pages of data.

```
const DEFAULT_QUERY = 'redux';
const DEFAULT_PAGE = 0;

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 these constants to add the page parameter to your API request.

```
var url = `${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}\
`;

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

The `fetchSearchTopstories()` method will take the page as second argument. The `componentDidMount()` and `onSearchSubmit()` methods take the `DEFAULT_PAGE` for the initial API calls. They should fetch the first page on the first load. Every additional fetch should fetch the next page.

```
class App extends Component {

  ...

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

  fetchSearchTopstories(searchTerm, page) {
    fetch(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}\
```

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

```
${page}`)
      .then(response => response.json())
      .then(result => this.setSearchTopstories(result));
  }

  onSearchSubmit(event) {
    const { searchTerm } = this.state;
    this.fetchSearchTopstories(searchTerm, DEFAULT_PAGE);
    event.preventDefault();
  }

  ...

}
```

Now you can use the current page from the API response in `fetchSearchTopstories()`. You want to this method in a button. Let's use the Button to fetch more paginated data from the Hacker News API. You only need to define the `onClick()` function which takes the current search term and the current page + 1. The result will be the next page.

```
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>
    );
  }
}
```

Make sure to default to page `0` when there is no result.

There is one step missing. You fetch the next page of data, but it will overwrite your old data. You want to concatenate the old and new data. Let's adjust the functionality to add the new data rather than to overwrite it.

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

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

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

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

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`. It is the next page. The old hits are already stored in your state and thus can be used.

Third you don't want to overwrite the old hits. You can merge old and new hits from the recent API request. The merge of both lists can be done with the ES6 array spread operator.

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

You can make one last adjustment. When you try the "More" button it only fetches a few items. The API url can be extended to fetch more data with each request. Again you can add more composeable path constants.

```
const DEFAULT_QUERY = 'redux';
const DEFAULT_PAGE = 0;
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.

```
fetchSearchTopstories(searchTerm, page) {
  fetch(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}${\
page}&${PARAM_HPP}${DEFAULT_HPP}`)
    .then(response => response.json())
    .then(result => this.setSearchTopstories(result));
}
```

Afterwards the request to the Hacker News API fetches more data than before.

Exercises:

- experiment with the Hacker News API parameters[66]

[66] 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. If it is there, the cache is used. Otherwise an API request is made to fetch the data.

In order to have a client cache for each result, you have to store multiple `results` rather than one `result` in your internal 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 search term (key).

At the moment your result in the component state looks similar to the following:

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

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

```
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` which is used to store each result.

```
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`. But why don't we use the `searchTerm` in the first place? The `searchTerm` is a fluctuant variable, because it gets changed every time you type into the Search input field. However in the end you will need a non fluctuant variable. It determines the recent submitted search term to the API and can be used to retrieve the correct result from the map of results.

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

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

Now you have to adjust the functionality where the result is stored to the internal component state. It should store each result by `searchKey`.

```
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 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()`.

```
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 to allocate values dynamically in an object.

The upper part needs to object spread all other results by `searchKey` in the state. Otherwise you would lose all results you 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 search term from your map of results.

```
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">
          <Search
```

```
            value={searchTerm}
            onChange={this.onSearchChange}
            onSubmit={this.onSearchSubmit}
          >
            Search
          </Search>
        </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 you will need to pass the `searchKey` rather than the `searchTerm` to the "More" button. Otherwise your paginated fetch depends on the `searchTerm` value that is fluctuant. Moreover make sure to keep the fluctuant `searchTerm` property for the input field in the "Search" component.

The search functionality should work again. It stores all results from the Hacker News API.

Additionally the `onDismiss()` method needs to get improved. It still deals with the result object. Now it has to deal with multiple results.

```
  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 }
      }
    });
  }
```

The “Dismiss” button should work again.

However, nothing stops the app from sending an API request on each submit. Even though there might be already a result, there is no check that prevents the request. The cache functionality is not complete yet. The last step would be to prevent the call when a result is available in the state.

```
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, DEFAULT_PAGE);
    }

    event.preventDefault();
  }

  ...

}
```

Now your client makes a request to the API only once although you search for a search 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.

Your *src/App.js* should look like the following by now:

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

const DEFAULT_QUERY = 'redux';
const DEFAULT_PAGE = 0;
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=';

class App extends Component {

  constructor(props) {
    super(props);

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

    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);
  }

  componentDidMount() {
    const { searchTerm } = this.state;
    this.setState({ searchKey: searchTerm });
    this.fetchSearchTopstories(searchTerm, DEFAULT_PAGE);
  }

  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 }
      }
    });
  }

  fetchSearchTopstories(searchTerm, page) {
    fetch(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}\
${page}&${PARAM_HPP}${DEFAULT_HPP}`)
      .then(response => response.json())
      .then(result => this.setSearchTopstories(result));
  }

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

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

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

    if (this.needsToSearchTopstories(searchTerm)) {
      this.fetchSearchTopstories(searchTerm, DEFAULT_PAGE);
    }

    event.preventDefault();
```

```
  }

  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 }
      }
    });
  }

  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">
          <Search
            value={searchTerm}
            onChange={this.onSearchChange}
```

```
            onSubmit={this.onSearchSubmit}
          >
            Search
          </Search>
        </div>
        <Table
          list={list}
          onDismiss={this.onDismiss}
        />
        <div className="interactions">
          <Button onClick={() => this.fetchSearchTopstories(searchKey, page + 1)\
}>
            More
          </Button>
        </div>
      </div>
    );
  }
}

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

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

const Button = ({ onClick, className = '', children }) =>
  <button
    onClick={onClick}
    className={className}
    type="button"
  >
    {children}
  </button>

export default App;
```

You have learned to interact with an API in React! Let's recap the last chapters:

- React
  - ES6 class component lifecycle methods for different use cases
  - componentDidMount() for API interactions
  - conditional renderings
  - synthetic events on forms
- ES6

    - template strings
    - spread operator
    - computed property names
- General
    - Hacker News API interaction
    - native fetch browser API
    - client- and server-side search
    - pagination of data
    - client-side caching

Again it makes sense to take a break. Internalize the learnings and apply them on your own. You can experiment with the source code you have written so far.

Advanced React Components

The chapter will focus on the implementation of advanced React components. Before you jump into this, you will need to know how to test your components. Afterwards you are prepared to implement your own higher order components and advanced interactions in React.

Snapshot Tests with Jest

Jest[67] is a JavaScript testing framework. At Facebook it is used to validate the JavaScript code. In the React community it is used for React components test coverage. Fortunately create-react-app already comes with Jest.

Let's start to test your first components. Before you can do that, you have to export the components from your *App.js* file to test them during the chapter.

```
...

class App extends Component {
  ...
}

...

export default App;

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

In your *App.test.js* file you will find a first test. It verifies that the App component renders without any errors.

```
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);
});
```

You can run it by using the interactive create-react-app scripts on the command line.

[67] https://facebook.github.io/jest/

```
npm run test
```

Now Jest enables you to write Snapshot tests. These tests make a snapshot of your rendered component and run this snapshot against future snapshots. When a future snapshot changes you will get notified during the test. You can either accept the snapshot change, because you changed the component implementation on purpose, or deny the change and investigate for an error.

Jest stores the snapshots in a folder. Only that way it can show the diff to future snapshots. Additionally the snapshots can be shared across teams.

You have to install an utility library before you can write your first Snapshot test.

```
npm install --save-dev react-test-renderer
```

Now you can extend the App component test with your first Snapshot test.

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

describe('App', () => {

  it('renders', () => {
    const div = document.createElement('div');
    ReactDOM.render(<App />, div);
  });

  test('snapshots', () => {
    const component = renderer.create(
      <App />
    );
    let tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });

});
```

Run your tests again and see how the tests either succeed or fail. They should succeed. 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.

Let's add more tests for our independent components. First the Search component.

```
import React from 'react';
import ReactDOM from 'react-dom';
import renderer from 'react-test-renderer';
import App, { Search } from './App';

...

describe('Search', () => {

  it('renders', () => {
    const div = document.createElement('div');
    ReactDOM.render(<Search>Search</Search>, div);
  });

  test('snapshots', () => {
    const component = renderer.create(
      <Search>Search</Search>
    );
    let tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });

});
```

Second the Button component.

```
...
import App, { Search, Button } from './App';

...

describe('Button', () => {

  it('renders', () => {
    const div = document.createElement('div');
    ReactDOM.render(<Button>Give Me More</Button>, div);
  });

  test('snapshots', () => {
    const component = renderer.create(
      <Button>Give Me More</Button>
    );
```

```
    let tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });

});
```

Last but not least for the Table component.

```
...
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', () => {
    const div = document.createElement('div');
    ReactDOM.render(<Table { ...props } />, div);
  });

  test('snapshots', () => {
    const component = renderer.create(
      <Table { ...props } />
    );
    let tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });

});
```

Snapshot tests usually stay pretty basic. You only want to cover that the component doesn't change its output.

Exercises:

- see how the Snapshot tests fail once you change your component implementation

    - either accept or deny the snapshot change
- keep your snapshots tests up to date when the implementation changes in future chapters
- read more about Jest in React[68]
- read more about ES6 export[69] and import[70]

[68] https://facebook.github.io/jest/docs/tutorial-react.html
[69] https://developer.mozilla.org/en/docs/web/javascript/reference/statements/export
[70] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Statements/import

Unit Tests with Enzyme

Enzyme[71] is a testing utility by Airbnb to assert, manipulate and traverse your React components. You can use it to conduct unit tests to complement your snapshot tests.

Let's see how you can use enzyme. First you have to install it since it doesn't come with create-react-app.

```
npm install --save-dev enzyme react-addons-test-utils
```

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 has two items.

```
import React from 'react';
import ReactDOM from 'react-dom';
import renderer from 'react-test-renderer';
import { shallow } from 'enzyme';
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('shows two items in list', () => {
    const element = shallow(
      <Table { ...props } />
    );

    expect(element.find('.table-row').length).toBe(2);
  });

});
```

[71] https://github.com/airbnb/enzyme

Shallow renders the component without child components. You can make the test very dedicated to one component.

Enzyme has overall three rendering mechanisms in its API. You already know `shallow()`, but there also exist `mount()` and `render()`. Both instantiate instances of the parent component and all child components. Additionally `mount()` gives you more access to the component lifecycle methods. But when to use which render mechanism? Here some rules of thumb:

- 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 children rendering with less overhead than `mount()` and you are not interested in lifecycle methods, use `render()`

You could continue to unit test your components. But make sure to keep the tests simple and maintainable. Otherwise you will have to refactor them once you change your components. That's why Facebook introduced Snapshot tests with Jest in the first place.

Exercises:

- keep your unit tests up to date during the following chapters
- read more about enzyme and its rendering API[72]

[72] https://github.com/airbnb/enzyme

Loading ...

Now let's get back to the application. You might want to show a loading indicator when you submit a search request to the Hacker News API. The request is asynchronous and you should show your user some feedback that something is about to happen. Let's define a reusable Loading component in your *App.js* file.

```
const Loading = () =>
  <div>Loading ...</div>
```

Now you will need a property to store the loading state. Based on the loading state you can decide to show the Loading component later on.

```
class App extends Component {

  constructor(props) {
    super(props);

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

    ...
  }

  ...

}
```

The initial value of that property is false. You don't load anything before the App component is mounted.

When you make the request, you set a loading state to true. Eventually the request will succeed and you can set the loading state to false.

```
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 }
      },
      isLoading: false
    });
  }

  fetchSearchTopstories(searchTerm, page) {
    this.setState({ isLoading: true });

    fetch(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}\
${page}&${PARAM_HPP}${DEFAULT_HPP}`)
      .then(response => response.json())
      .then(result => this.setSearchTopstories(result));
  }

  ...

}
```

In the last step you will use the Loading component in your App. A conditional rendering based on the loading state will decide whether you show a Loading or Button component. The latter one is your button to fetch more data.

```
class App extends Component {

  ...

  render() {
    const {
      searchTerm,
      results,
      searchKey,
      isLoading
    } = 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">
          { isLoading
            ? <Loading />
            : <Button
              onClick={() => this.fetchSearchTopstories(searchKey, page + 1)}>
              More
            </Button>
          }
        </div>
      </div>
    );
  }
}
```

Initially the Loading component will show up when you start your app, 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. The "More" button to fetch more data appears. Once you fetch more data, the button will disappear. Instead the Loading component will show up.

Exercises:

- use a library like Font Awesome[73] to show a loading icon instead of text

[73] http://fontawesome.io/

Higher Order Components

Higher order components (HOC) are an advanced concept in React. HOCs are equivalent to higher order functions. They take any input - most of the time a component - and return a component as output. The returned component is an enhanced version.

Let's do a simple HOC which takes a component as input and returns a component. You can place it in your *App.js* file.

```
const withSomething = (Component) => (props) =>
  <Component { ...props } />
```

In the example the input component would stay the same as the output component. It renders the same component instance and passes all of the props to the output component.

Now let's enhance the output component. The output component should show the Loading component, when the loading state is true, otherwise it should show the input component.

```
const withLoading = (Component) => ({ isLoading, ...rest }) =>
  isLoading ? <Loading /> : <Component { ...rest } />
```

You use a conditional rendering based on the loading property. It will return the Loading component or input component.

Additionally you may have noticed the `{ isLoading, ...rest }` ES6 rest destructuring. It takes one property out of the object, but keeps the remaining object. It works with multiple properties as well. You might have already read about it in destructuring assignment[74].

In general it can be very efficient to spread an object as input for a component. See the difference in the following code snippet.

```
// before you would have to destructure the props before passing them
const { foo, bar } = props;
<SomeComponent foo={foo} bar={bar} />

// but you can use the object spread operator to pass all object properties
<SomeComponent { ...props } />
```

Now you can use the HOC. In your use case you want to show either the "More" button or Loading component. The HOC can take the Button component as input. The enhanced output component is a ButtonWithLoading component.

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

```
const Button = ({ onClick, className = '', children }) =>
  <button
    onClick={onClick}
    className={className}
    type="button"
  >
    {children}
  </button>

const Loading = () =>
  <div>Loading ...</div>

const withLoading = (Component) => ({ isLoading, ...rest }) =>
  isLoading ? <Loading /> : <Component { ...rest } />

const ButtonWithLoading = withLoading(Button);
```

Everything is declared. As the 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.

```
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>
    );
  }
}
```

Higher order components are an advanced technique in React. They have multiple purposes like improved reusability of components, greater abstraction, composability of components and manipulations of props, state and view.

Exercises:

- experiment with the HOC you have created
- optionally implement another HOC

Advanced Sorting

You have already implemented a client- and server-side search interaction. Since you have a Table component, it would make sense to enhance the Table with advanced interactions. What about enabling sorting by the Table columns?

It would be possible to write your own sort function, but personally I prefer to use a utility library for such cases. Lodash[75] is one of these utility libraries. Let's install and use it for the sort functionality.

```
npm install --save lodash
```

Now you can import the sort functionality of lodash in your *App.js* file.

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

You have several columns in your Table. Among these you have title, author, comments and points columns. You can define sort functions where each function takes a list and returns a list of items sorted by property. Additionally you will need one default sort function which doesn't sort but only returns the unsorted list.

```
...

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 {
  ...
}
...
```

You can see that two of the sort functions return a reversed list. That's because you want to see the items with the highest comments and points rather than to see the items with the lowest.

The `SORTS` object allows you to reference any sort function now.

Again your App component is responsible for storing the state of the sort. The initial state will be the initial default sort function, which doesn't sort at all and returns the input list as output.

[75] https://lodash.com/

```
this.state = {
  results: null,
  searchKey: '',
  searchTerm: DEFAULT_QUERY,
  isLoading: false,
  sortKey: 'NONE',
};
```

Once you choose a different `sortKey`, let's say `AUTHOR`, you will sort the list by the author property.

Now you can define a new sort method in your App component that simply sets a `sortKey` to your internal component state.

```
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 your Table component.

```
class App extends Component {

  ...

  render() {
    const {
      searchTerm,
      results,
      searchKey,
      isLoading,
      sortKey
    } = 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">
          <Search
            value={searchTerm}
            onChange={this.onSearchChange}
            onSubmit={this.onSearchSubmit}
          >
            Search
          </Search>
        </div>
        <Table
          list={list}
          sortKey={sortKey}
          onSort={this.onSort}
          onDismiss={this.onDismiss}
        />
```

```
        <div className="interactions">
          <ButtonWithLoading
            isLoading={isLoading}
            onClick={() => this.fetchSearchTopstories(searchKey, page + 1)}>
            More
          </ButtonWithLoading>
        </div>
      </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. Afterwards it still maps over the sorted list.

```
const Table = ({
  list,
  sortKey,
  onSort,
  onDismiss
}) =>
  <div className="table">
    { SORTS[sortKey](list).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>
```

In theory the list would get sorted by one of the functions. But the default sort is set to `NONE`. So far no one executes the `onSort()` method to change the `sortKey`. Let's extend the Table with a header row that uses Sort components in columns to sort each column.

```
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.

```
const Sort = ({ sortKey, onSort, children }) =>
  <Button onClick={() => onSort(sortKey)}>
    {children}
  </Button>
```

As you can see, the Sort component reuses your common Button component. On click, each individual passed `sortKey` will get set by the `onSort()` method. Now you should be able to sort the list when you click on the column headers.

But a button in a column header looks a bit silly. Let's give the Sort a proper `className`.

```
const Sort = ({ sortKey, onSort, children }) =>
  <Button
    onClick={() => onSort(sortKey)}
    className="button-inline"
  >
    {children}
  </Button>
```

It should look nice now. The next goal would be to implement reverse sort as well. The list should get reverse sorted once you click a Sort component twice. First you need to define the reverse state.

```
this.state = {
  results: null,
  searchKey: '',
  searchTerm: DEFAULT_QUERY,
  isLoading: false,
  sortKey: 'NONE',
  isSortReverse: false,
};
```

Now in your sort method you can evaluate if the list is reverse sorted. It is when `sortKey` in the state is the same as the incoming `sortKey` and the reverse state is not already set to true.

```
onSort(sortKey) {
  const isSortReverse = this.state.sortKey === sortKey && !this.state.isSortReve\
rse;
  this.setState({ sortKey, isSortReverse });
}
```

Again you can pass the reverse prop to your Table component.

```
class App extends Component {

  ...

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

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

    const list = (
      results &&
```

```
      results[searchKey] &&
      results[searchKey].hits
    ) || [];

    return (
      <div className="page">
        ...
        <Table
          list={list}
          sortKey={sortKey}
          isSortReverse={isSortReverse}
          onDismiss={this.onDismiss}
          onSort={this.onSort}
        />
        ...
      </div>
    );
  }
}
```

The Table has to have an arrow function block body to compute the data now.

```
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>
  );
}
```

The reverse sort should work now.

Last but not least you have to deal with one open question for the sake of an improved user experience. Can a user distinguish which column is actively sorted? So far it is not possible. Let's give the user a visual feedback. Each Sort component gets its specific `sortKey` already. It could be used to identify the activated sort. You can pass the `sortKey` from the internal component state as active sort key to your Sort component.

```
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 style={{ width: '10%' }}>
            Archive
          </span>
        </span>
      </div>
      { reverseSortedList.map(item =>
          ...
      )}
    </div>
  );
}
```

Now in your Sort component you know based on the `sortKey` and `activeSortKey` if the sort is active. Give your Sort component an extra `class` attribute, when it is sorted, to give the user a visual feedback.

```
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>
  );
}
```

The way to define the `class` is a bit clumsy, isn't it? There is a neat little library to get rid of this. First you have to install it.

```
npm install --save classnames
```

And second you have to import it on top of your *App.js* file.

```
import React, { Component } from 'react';
import { sortBy } from 'lodash';
import classNames from 'classnames';
import './App.css';
```

Now you can use it to define your component `className` with conditional classes.

```
const Sort = ({
  sortKey,
  activeSortKey,
  onSort,
  children
}) => {
  const sortClass = classNames(
    'button-inline',
    { 'button-active': sortKey === activeSortKey }
  );

  return (
    <Button
      onClick={() => onSort(sortKey)}
      className={sortClass}
    >
      {children}
    </Button>
  );
}
```

Your advanced sort interaction is complete now.

Exercises:

- use a library like Font Awesome[76] to indicate the (reverse) sort
  - it could be an arrow up or down icon next to each Sort header
- read more about the classnames library[77]

[76] http://fontawesome.io/
[77] https://github.com/JedWatson/classnames

Lift State

Only the App component is a stateful ES6 component in your application. It handles a lot of application state and logic (methods). Maybe you have noticed that you pass a lot of properties to your Table component. Most of the props are only used in the component. It makes no sense that the App component knows about them.

The sort functionality is only handled in the Table component. You could move it closer to the Table component. The App component doesn't need to know about it at all. The process of refactoring substate from one component to another is known as *lifting state*. In your case you want to move state that isn't used in the App component closer to the Table component. The state moves down from parent to child component.

In order to deal with state and methods in the Table component, it has to become an ES6 class component. The refactoring from functional stateless component to ES6 class component is straight forward.

Your Table component as functional stateless component:

```
const Table = ({
  list,
  sortKey,
  isSortReverse,
  onSort,
  onDismiss
}) => {
  const sortedList = SORTS[sortKey](list);
  const reverseSortedList = isSortReverse
    ? sortedList.reverse()
    : sortedList;

  return(
    ...
  );
}
```

Can get refactored to an ES6 class component:

```
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.

```
class Table extends Component {

  constructor(props) {
    super(props);

    this.state = {};
  }

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

Now you can move state and methods from your App component down to your Table component.

```
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.isSortRe\
verse;
    this.setState({ sortKey, isSortReverse });
  }

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

Don't forget to remove the moved state and `onSort()` method from your App component.

```
class App extends Component {

  constructor(props) {
    super(props);

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

  ...

}
```

Additionally you can make the Table component API more lightweight. Remove the props which are handled internally in the Table component.

```
class App extends Component {

  ...

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

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

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

    return (
      <div className="page">
        ...
        <Table
          list={list}
          onDismiss={this.onDismiss}
        />
```

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

Now in your Table component you can use the internal `onSort()` method and the internal Table state.

```
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>
    );
  }
}
```

Your application should still work. But you made a crucial refactoring. You moved functionality and state closer to a component. Other components got more lightweight again. Additionally the component API of the Table got more lightweight because it deals internally with the sort functionality.

The process of lifting state can go the other way as well: from child to parent component - lifting state up. Imagine you were dealing with internal state in a child component. Now you want to fulfill a requirement to show the state in your parent component as well. You would have to lift up the state to your parent component. But it goes even further. Imagine you want to show the state in a sibling component of your child component. Again you would have to lift the state up to your parent component. The parent component deals with the internal state, but exposes it to both child components.

Exercises:

- read more about React Lift State[78]

[78] https://facebook.github.io/react/docs/lifting-state-up.html

Your *src/App.js* should look like the following by now:

```
import React, { Component } from 'react';
import { sortBy } from 'lodash';
import classNames from 'classnames';
import './App.css';

const DEFAULT_QUERY = 'redux';
const DEFAULT_PAGE = 0;
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=';

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 {

  constructor(props) {
    super(props);

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

    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);
```

```
  }

  componentDidMount() {
    const { searchTerm } = this.state;
    this.setState({ searchKey: searchTerm });
    this.fetchSearchTopstories(searchTerm, DEFAULT_PAGE);
  }

  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
    });
  }

  fetchSearchTopstories(searchTerm, page) {
    this.setState({ isLoading: true });

    fetch(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}\
${page}&${PARAM_HPP}${DEFAULT_HPP}`)
      .then(response => response.json())
      .then(result => this.setSearchTopstories(result));
  }

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

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

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

    if (this.needsToSearchTopstories(searchTerm)) {
      this.fetchSearchTopstories(searchTerm, DEFAULT_PAGE);
    }

    event.preventDefault();
  }

  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 }
      }
    });
  }

  render() {
    const {
      searchTerm,
      results,
      searchKey,
      isLoading
    } = 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">
          <Search
            value={searchTerm}
            onChange={this.onSearchChange}
            onSubmit={this.onSearchSubmit}
          >
            Search
          </Search>
        </div>
        <Table
          list={list}
          onDismiss={this.onDismiss}
        />
        <div className="interactions">
          <ButtonWithLoading
            isLoading={isLoading}
            onClick={() => this.fetchSearchTopstories(searchKey, page + 1)}>
            More
          </ButtonWithLoading>
        </div>
      </div>
    );
  }
}

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

```
    <form onSubmit={onSubmit}>
      <input
        type="text"
        value={value}
        onChange={onChange}
      />
      <button type="submit">
        {children}
      </button>
    </form>

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.isSortRe\
verse;
    this.setState({ sortKey, isSortReverse });
  }

  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 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>
      );
    }
}

const Sort = ({
  sortKey,
  activeSortKey,
  onSort,
  children
}) => {
  const sortClass = classNames(
    'button-inline',
```

```
    { 'button-active': sortKey === activeSortKey }
  );

  return (
    <Button
      onClick={() => onSort(sortKey)}
      className={sortClass}
    >
      {children}
    </Button>
  );
}

const Button = ({ onClick, className = '', children }) =>
  <button
    onClick={onClick}
    className={className}
    type="button"
  >
    {children}
  </button>

const Loading = () =>
  <div>Loading ...</div>

const withLoading = (Component) => ({ isLoading, ...rest }) =>
  isLoading ? <Loading /> : <Component { ...rest } />

const ButtonWithLoading = withLoading(Button);

export default App;

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

You have learned advanced component techniques in React! Let's recap the last chapters:

- React
  - Jest allows you to write snapshot tests for your components

  - Enzyme allows you to write unit tests for your components
  - higher order components are a common way to build advanced components
  - implementation of advanced interactions in React
  - conditional classNames with a neat helper library
  - lift state up and down
- ES6
  - rest destructuring

Deploy your App to Production

In the end no app should stay on localhost. You want to go live! Heroku is a platform as a service where you can host your app. They offer a seamless integration with React. To be more specific: It's possible to deploy a create-react-app in minutes. It's a zero-configuration deployment which follows the philosophy of create-react-app.

You need to fulfill two requirements before you can deploy your app to Heroku:

- install the Heroku CLI[79]
- create a free Heroku account[80]

If you have installed Homebrew, you can install the Heroku CLI from command line:

```
brew update
brew install heroku-toolbelt
```

Now you can use git and Heroku CLI to deploy your app.

```
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 it. I hope your app is up and running now. If you run into problems you can check the following resources:

- Deploying React with Zero Configuration[81]
- Heroku Buildpack for create-react-app[82]

[79] https://devcenter.heroku.com/articles/heroku-command-line
[80] https://www.heroku.com/
[81] https://blog.heroku.com/deploying-react-with-zero-configuration
[82] https://github.com/mars/create-react-app-buildpack

Final Words

That was the last chapter of the book. I hope you liked it so far. Share it with your friends, it would mean a lot to me.

But where can you go from here? You can either extend the application on your own or dive into your own React project. Before you dive into another book or tutorial, you should create your own hands-on React project. Do it for one week, release it somewhere and reach out to me on Twitter[83]. I am curious what you will build after you have read the book. You can also find me on GitHub[84] to share your repository.

If you are looking for further extensions for your app, I can recommend two paths:

- implement routing for your App with react-router
- use a state management library (Redux or MobX) for scaling state management

After you have built your own React application, you may want to experiment with Redux. Redux is a state management library. You can build your own SoundCloud Client in React + Redux[85] in a tutorial. The tutorial shows you more advanced tooling around React. It has multiple extensions to cover topics like ESLint, Flow, Normalizr, MobX and Observables[86].

More Readings:

- Reasons why I moved from Angular to React[87]
- Tips to learn React + Redux[88]
- Redux or MobX: An attempt to dissolve the Confusion[89]

More Resources:

- A real SoundCloud Client in React + Redux: Live[90] & Source[91]
- A real SoundCloud Client in React + MobX: Source[92]

[83] https://twitter.com/rwieruch
[84] https://github.com/rwieruch
[85] https://www.robinwieruch.de/the-soundcloud-client-in-react-redux
[86] https://github.com/rwieruch/react-redux-soundcloud
[87] https://www.robinwieruch.de/reasons-why-i-moved-from-angular-to-react/
[88] https://www.robinwieruch.de/tips-to-learn-react-redux/
[89] https://www.robinwieruch.de/redux-mobx-confusion/
[90] http://www.favesound.de/
[91] https://github.com/rwieruch/favesound-redux
[92] https://github.com/rwieruch/favesound-mobx

In general I invite you to visit my website www.robinwieruch.de[93] to find more interesting topics. You can subscribe[94] to my Newsletter to get content updates.

If you liked the book, I hope you will share it with your friends. I would really appreciate it. Thanks for reading - Robin.

[93] https://www.robinwieruch.de/
[94] http://eepurl.com/caLPjr