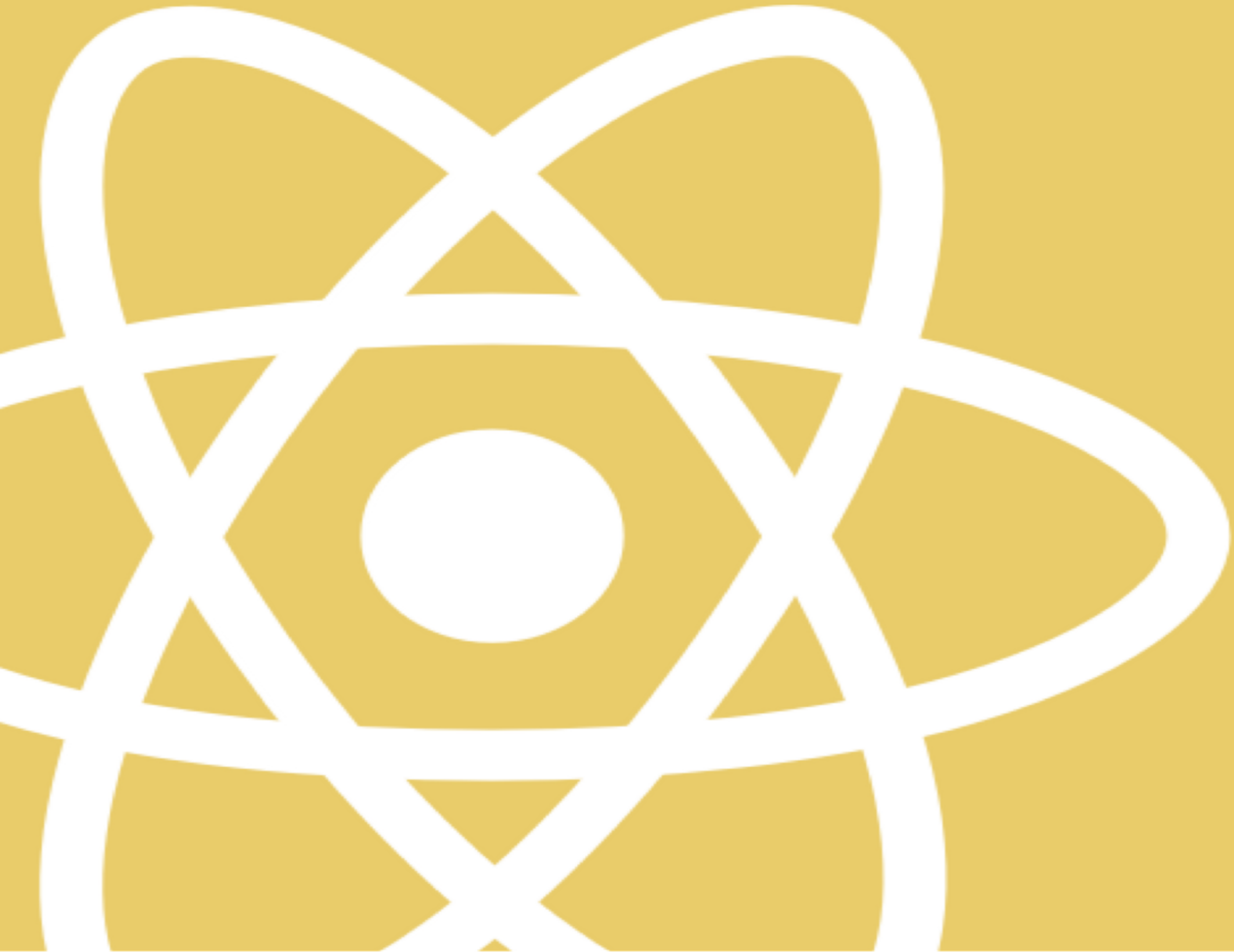
by robin wieruch

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 2016-12-13

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 larger 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 honest 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 pretty quickly into Redux. Still it helped a lot of people to get started. In my opinion it's a great tutorial to get a bigger picture of React + Redux. I use every free minute to improve the material, but it's time consuming. I am going to overhaul it as a whole in the future.

In the Road to learn React[2] I want to offer a foundation before you start to dive into the more advanced React ecosystem. It has less tooling and less external state management, but more React. It explains general concepts and patterns. Additionally it links to the official documentation, because it's a great reference to learn React. After all I want to give a clear road to learn the React ecosystem. It should provide you a solid foundation before you dive into more advanced topics like Redux.

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

[2]https://leanpub.com/the-road-to-learn-react

FAQ

How do I get updates? You can subscribe[3] or follow me on Twitter[4] for updates. It keeps me motivated to work on the book as well. Once you have a copy of the book from Leanpub[5], the book will stay updated.

Does it cover Redux? So far 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 a presentable application.

But why is the book for free? I have put a lot of effort into this and will do so in the future. My desire is to reach as many people as possible. Everyone should be enabled to learn React. Still you can decide to pay something when you can effort it. It's pay as you want. Once you have your version of the book, you will automatically get all the updates. I would appreciate each of your social shares for the book.

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.

Will you add more chapters in the future? It depends on the community. If there is an acceptance for the book, I will deliver more chapters. Additionally I would love to hear your thoughts about possible chapters to improve and enrich the learning experience.

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

What you can expect (so far...)

- well rounded real world Hacker News client in React[7]
- no complicated configurations
- create-react-app to bootstrap your application
- efficient lightweight code
- only React setState as state management (so far...)
- JavaScript ES6 along the way
- the React API with setState and lifecycle methods
- a real world API (Hacker News)
- advanced interactions
  - client-sided sorting
  - client-sided filtering
  - server-sided searching
- 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

[7]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[8] or give me feedback on Twitter[9].

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[10]. It guides you to implement your own SoundCloud application with a state management library.

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

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

[10]http://www.robinwieruch.de/the-soundcloud-client-in-react-redux

Bootstrap your React App

The chapter will give you an introduction to React. It clarifies why you should learn React in the first place. Once that's clarified you will bootstrap your first React app. Along the way you will get an introduction to JSX to 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. You can get easily started to render your first components 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 you to build your view layer. It's a library and not a whole 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.

First you can learn them step by step without worrying to understand everything at once. It's different in comparison to a framework which gives you every building block from the start.

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

The first generation SPA frameworks arrived at an enterprise level. React stays innovative and gets adapted by multiple tech thought leader companies like Airbnb, Netflix and of course Facebook[11]. React is probably one of the best choices for building UI nowadays. It has a good design, an amazing ecosystem and a great community. Everyone is keen to experience where it will lead us in 2017.

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

Requirements

Before you start to read the book, you should be quite familiar with HTML, CSS and JavaScript (ES5). Additionally you will need a working editor and terminal[12]. Last but not least you will need an installation of node and npm[13].

These are my versions of node and npm at the time of writing the book.

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

[12]http://www.robinwieruch.de/developer-setup/
[13]https://nodejs.org/en/

create-react-app

You will use create-react-app[14] to bootstrap your app. It's an opinionated but zero-configuration starter kit for React introduced by Facebook. People like and would recommend it to starters by 96%[15]. In create-react-app the tools 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 on your command line. You should already have npm installed to install the package.

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

Now you can bootstrap your first app and navigate into the directory:

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

When you open the app in your editor, you will find the following folder structure:

```
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
```

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. But there is also the *App.test.js* for tests and the *index.js* as entry point to the React world. We will get to know them in a later chapter.

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

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

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

// Runs the tests
npm test

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

You can read more about the scripts and create-react-app[16] 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

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

Introduction to JSX

Now you will get to know JSX. Let's dive into the source code provided by create-react-app. 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;
```

create-react-app already scaffolded a boilerplate application. In the file you have an ES6 class component with the name App. Basically you can use the `<App />` component everywhere in your application now. Once you use it, it will produce an instance of your component. The elements it returns are specified in the `render()` function.

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 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 in the beginning 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.

Try it. Define a property and use it in JSX.

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

class App extends Component {
  render() {
    const helloWorld = 'Welcome to React';
    return (
      <div className="App">
        <div className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <h2>{helloWorld}</h2>
        </div>
        <p className="App-intro">
          To get started, edit <code>src/App.js</code> and save to reload.
        </p>
      </div>
    );
  }
}

export default App;
```

Additionally you might have noticed the `className` attribute. Because of technical reasons JSX had to replace HTML attributes like class (className) and for (htmlFor).

ES6 Sugar:

In ES5 you declare variables with var. In ES6 there are two more variable declarations: const and let.

Exercises:

- read more about JSX[17]
- read more about React components, elements and instances[18]
- read more about supported HTML attributes in React[19]
- read more about ES6 const[20] and let[21] variable declarations
- define more variables to render them in your JSX

[17]https://facebook.github.io/react/docs/introducing-jsx.html
[18]https://facebook.github.io/react/blog/2015/12/18/react-components-elements-and-instances.html
[19]https://facebook.github.io/react/docs/dom-elements.html
[20]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const
[21]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's 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 React components. That's how you can easily integrate React in every app.

ReactDOM.render expects two arguments.

The first argument is JSX. It already takes your App component, but you don't need to pass a component. It would be sufficient to use `<div>Hello React World</div>`. After all the first argument is the content to be 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.

Exercises:

- open your *public/index.html* file to find the `id="root"`
- read more about React rendering element[22]

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

Map in JSX

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

Since you can use JavaSript in JSX, it's possible to map over your data to display each item.

```
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(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;
```

Additionally you have to assign a key property to each list element. Only that way React is able to identify added, changed and removed items.

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

Give your elements a stable id. Don't make the mistake to use the array key which isn't stable. React will have a hard time to identify the items properly when the order of them changes.

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

Now when you open your app in a browser both list items should get rendered and be visible.

ES6 Sugar:

Let's have a look again at the map function. It takes a function itself, which you can write more concise in ES6.

First you can use the 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>
  );
})}
```

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

```
{ 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>
)}
```

Now your JSX looks more readable again.

Exercises:

- read more about React lists and keys[23]
- read more about ES6 arrow functions[24]
- make yourself comfortable with standard built-in functionalities in JavaScript[25]
- use more JavaScript expression on your own in JSX (e.g. ternary[26])

[23]https://facebook.github.io/react/docs/lists-and-keys.html
[24]https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/Arrow_functions
[25]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map
[26]https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Conditional_Operator

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
  - ReactDOM.render() is an entry point for a React app
  - built-in JavaScript functionalities like map can be used render a list of items
- ES6
  - more variable declarations with const and let
  - arrow functions with block and concise bodies 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 focus on the basics in React. You will learn to manage internal component state, to implement component interactions and to compose components. Moreover it will show you to declare components in different ways.

Internal Component State

Internal component state allows you to store, modify and delete properties of your components. 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.

Notice that you have to call `super(props);` to call the constructor of the parent class. It's mandatory.

```
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>
    );
  }
}
```

In the render function you have access to the component internal state by using `this`. As before you can map over the list of items.

ES6 Sugar:

You can use a shorthand syntax to initialize properties in an object. In your case it makes sense because the property and variable share the same name.

```
// instead of
this.state = {
  list: list
};

// you can use
this.state = {
  list
};
```

Exercises:

- read more about the ES6 class constructor[27]

[27]https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes#Constructor

Interactions in Components

Now you have some static internal state in your component. But you don't manipulate the internal state so far. The best to experience state manipulation is by implementing a component interaction. It could be a search field. The input of the search field should be used to filter your list.

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 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>
    );
  }
}
```

In the following scenario you will type into the field and filter the list by the search term. To be able to filter the list, you need the value of the input field. But where to access the value? Naturally in React the value goes into the internal component state to be accessible und updatable.

Let's define a callback function for the input field which manipulates the internal component state.

```
class App extends Component {

  ...

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

The callback function is bound to the component and thus a component method. 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);
  }

  onSearchChange(event) {
    ...
  }

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

The callback gives you access to the input field event. The event has the value of the input field in its target object. By using `this.setState` you can manipulate the internal component state. Additionally you should define the initial state for the query.

```
class App extends Component {

  constructor(props) {
    super(props);

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

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

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

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

Now you store the value to your component state every time the value in the input field changes. But one piece is missing.

In React applications you will strictly follow an unidirectional data flow. The input field already updated the internal component state, but to finish the roundtrip you have to update the value of the input field with the internal state too.

You have to retrieve the query property from the internal component state and set it as value in the input field.

```
class App extends Component {

  ...

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

The unidirectional data flow roundtrip finished. Now the input field value is under your control.

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. So what's next? You save your input value as query, but what about searching the list?

Before you map the list you can filter it.

```
class App extends Component {

  ...

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

But now you need to pass the query property to your evaluation process. That's why you can use a higher order function, which takes the query but returns another function. The returned function will do the evaluation for each item, but has access to the query property.

```
function isSearched(query) {
  return function(item) {
    return !query || item.title.toLowerCase().indexOf(query.toLowerCase()) !== -\
1;
  }
}

class App extends Component {

  ...

}
```

You filter the list only when a query is set. When a query is set, you match the incoming query pattern with the title of the item. Only when the pattern matches you return true. Don't forget to lower case everything, otherwise there will be mismatches between a query 'redux' and a item title 'Redux'.

The search field should do its work now. Try it.

ES6 Sugar:

You can even add some more ES6 syntax again. First you can destructure objects and arrays. Try it by destructuring the internal component state.

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

Second you can make the function more concise in ES6 by using arrow functions again.

You can refactor your higher order function:

```
// before
function isSearched(query) {
  return function(item) {
    return !query || item.title.toLowerCase().indexOf(query.toLowerCase()) !== -\
1;
  }
}

// after
const isSearched = (query) => (item) => !query || item.title.toLowerCase().index\
Of(query.toLowerCase()) !== -1;
```

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 quite often that you will use a function which returns a function. These are called higher order functions. In ES6 you can express these more concise with arrow functions.

Exercises:

- read more about React forms[28]
- read more about React events[29]
- read more about ES6 destructuring[30]
- read more about higher order functions[31]

[28]https://facebook.github.io/react/docs/forms.html
[29]https://facebook.github.io/react/docs/handling-events.html
[30]https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment
[31]https://en.wikipedia.org/wiki/Higher-order_function

Composable Components

So far you have one large App component. Let's start to split it up into smaller composable components. You can define a component for the search input and a component for the list items.

```
class App extends Component {

  ...

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

class Search extends Component {

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

}

class Table extends Component {

  render() {
    const { list, pattern } = 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>
        </div>
      )}
      </div>
    );
  }
}
```

Properties, they are called props in React, can be passed to components. The components themselves have every property accessible in the `props` object.

Now you have three ES6 class components.

Still you can't compose components into each other. That's why there exists the React children property. You can try the following to see the children property in action.

```
class App extends Component {

  ...

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

class Search extends Component {

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

  ...

}
```

The children property should get rendered in the Search component now.

By using the children property you can compose components into each other. You can pass whole components and component hierarchies as children.

Exercises:

- read more about the composition model of React[32]

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

Different Component Declarations

Now you have three ES6 class components. But you can do better by using functional stateless components. Before you will refactor your components to functional stateless components, let me explain the different types of component declarations.

Functional stateless components are functions which get an input and return an output. The output is a component instance. There are no side effects (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.

Besides of the functional stateless component, you already know the ES6 class component with the `this` object and two lifecycle methods.

Additionally there is a third type of component declaration: React.createClass. It was used in older versions of React, but is declared as deprecated now. I still wanted to mention it, in case you come across these declarations in older React material.

But when to use functional stateless components and ES6 class components? A good rule of thumb is to use functional stateless components when you don't need internal component state nor 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 will refactor it to an ES6 class component.

The App component uses internal state. That's why it has to stay as ES6 class component. But both of your new components are stateless without lifecycle methods. Let's refactor the Search component together to a stateless functional component. The Table 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} />
    </form>
  );
}
```

You already know and can apply the props destructuring. The best practice is 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 that arrow functions allow you to keep your functions concise. 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 is 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.

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

  // do something

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

Now you have one lightweight functional stateless component. 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 component to a stateless functional component
- read more about ES6 class components and functional stateless components[33]

[33]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, but feel free to use your own style. If you don't come up with your own style, copy and paste the following styles.

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;
  text-transform: uppercase;
}

.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 components. Don’t forget to use React className.

```
class App extends Component {

  ...

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

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

const Table = ({ list, pattern }) =>
  <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>
      </div>
    )}
  </div>
```

Another technique in React to style components is inline style. You can pass style objects to your components. Let's keep the Table column width flexible by using inline style.

```
const Table = ({ list, pattern }) =>
  <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: '15%' }}>
          {item.num_comments}
        </span>
        <span style={{ width: '15%' }}>
          {item.points}
        </span>
      </div>
    )}
  </div>
```

It's really inlined now. You could define the style objects outside as well.

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

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

const smallColumn = {
  width: '15%',
};
```

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. I don't want to be opinionated here, but I leave you some options to read about it. I'm open to your thoughts about other options as well.

- React Inline Style

- CSS Modules[34]
- styled-components[35]

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

[34]https://github.com/css-modules/css-modules
[35]https://github.com/styled-components/styled-components

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 = (query) => (item) => !query || item.title.toLowerCase().index\
Of(query.toLowerCase()) !== -1;

class App extends Component {

  constructor(props) {
    super(props);

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

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

  onSearchChange(event) {
    this.setState({ query: event.target.value });
  }
  render() {
    const { query, list } = this.state;
    return (
      <div className="page">
        <div className="interactions">
          <Search value={query} onChange={this.onSearchChange}>
            Search
          </Search>
        </div>
        <Table list={list} pattern={query} />
      </div>
    );
  }
}

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

const Table = ({ list, pattern }) =>
  <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: '15%' }}>
          {item.num_comments}
        </span>
        <span style={{ width: '15%' }}>
          {item.points}
        </span>
      </div>
    )}
  </div>

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
  - compose components with children
  - usage and implementation of ES6 class components and functional stateless components
  - approaches to style your components
- ES6
  - destructuring of objects and arrays
  - arrow functions with concise and block body
- General
  - higher order functions

Again it makes sense to make 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 could read more in the official documentation[36].

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

Getting Real with an API

Now it's time to get real with an API. Do you know Hacker News[37]? It's a great news aggregator. You will use the Hacker News API to fetch trending stories from the platform. There is a basic[38] and search[39] API. The latter one makes sense in your case to search stories on Hacker News. You could visit the API specification[40] to get a glimpse on the data structure.

[37]https://news.ycombinator.com/

[38]https://github.com/HackerNews/API

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

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

Lifecycle Methods

But the basics come first. You will need the knowledge about React lifecycle methods before you can start. These methods are a hook into the lifecycle of a React component which you can overwrite. There are only 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 props or state of a component change the render method is called.

Now you know two lifecycle methods and when they are called. But there are more.

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

The mounting process has 4 lifecycle methods.

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

The componentDidMount method is usually used to get data from an API endpoint. But what about the update lifecycle of a component? Overall it has five lifecycle methods.

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

You don't need to know all of them from the beginning. But still it's good to know that each lifecycle method can be used for a specific use case. For instance shouldComponentUpdate() is mostly used in a mature React app to prevent a component to update 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 which has only one lifecycle method called componentWillUnmount. It's used to cleanup when you are about to destroy your component.

You only used the constructor and render method by now. These are the common used lifecycle methods for ES6 class components. Every other lifecycle method is used for more advanced use cases.

Exercises:

- read more about lifecycle methods in React[41]
- read more about the state and lifecycle in React[42]

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

Fetch Data

Now you are prepared to fetch data from the Hacker News API. I mentioned one lifecycle method which can be used to fetch data.

Let's setup the path constants and default parameters to break the API request into smaller pieces.

```
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=';

...
```

By using ES6 template strings[43] you can concatenate the url.

```
`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${DEFAULT_QUERY}`

// 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.

```
...

class App extends Component {

  constructor(props) {
    super(props);

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

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

```
    this.setSearchTopstories = this.setSearchTopstories.bind(this);
    this.fetchSearchTopstories = this.fetchSearchTopstories.bind(this);
    this.onSearchChange = this.onSearchChange.bind(this);
  }

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

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

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

  ...
}
```

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

First you can remove the hard coded list of items, because you return a result from the Hacker News API. Now the initial state of your component has a empty result and default query. The same default query 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 mounted. In the very first fetch the default query from the component state is used.

Third the native fetch API is used. ES6 string template strings allow it to compose the path with the query. The response needs to get transformed and can finally be set in the internal component state. Be careful and don't forget to bind your new component methods.

Now you can use the fetched data instead of the hard coded list of items. But be careful. The result is not only a list of data. It's a complex object with meta information and a list of hits.[44] You can output the internal state with `console.log(this.state);` to visualize it.

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

```
class App extends Component {

  ...

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

Have you noticed the conditional rendering? Before you fetch the data from the Hacker News API, the result in your internal component state is null. Thus you can't render the Table. After that you fetch the data to fill your table with items.

Let's recap what happens during the component lifecycle. Your component gets initialized by the constructor. After that it renders the first time. Only after the the component rendered after the initialization, the componentDidMount lifecycle method runs. In that method you fetch the 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 again the render method, but this time with populated data in your internal component state. The Table gets rendered.

Let's get back to the conditional rendering. There is another approach when you return null for one of the blocks.

```
{ result && <Table list={result.hits} pattern={query} /> }
```

At the end it's a personal preference. But after all you should be able to see the fetched data in your list.

Exercises:

- read more about ES6 template strings[45]
- read more about React conditional rendering[46]

[45]https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Template_literals
[46]https://facebook.github.io/react/docs/conditional-rendering.html

- read more about the native fetch API[47]
- experiment with the Hacker News API[48]

[47]https://developer.mozilla.org/en/docs/Web/API/Fetch_API
[48]https://hn.algolia.com/api

Client- or Server-side Interaction: Search

When you use the search 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 query parameter.

You can define an `onSubmit` function in your ES6 class component, which fetches results from the Hacker News API like in your `componentDidMount()` lifecycle method. But it fetches it with the modified query. Don't forget to bind the function.

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      result: null,
      query: 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);
  }

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

  ...
}
```

The Search component gets extended by a button element. The button will explicit trigger the search. Otherwise you would hit the Hacker News API every time your input changes. As alternative you could debounce (delay) the `onChange` callback and spare the button, but it would add more complexity at this time. Let's keep it without debounce.

```
class App extends Component {

  ...

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

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

Additionally the Search component gets the `onSubmit` callback. The form uses the callback, but the button has to define itself as type of submit.

In the Table you can remove the filter functionality, because there will be no client-side filter anymore. The result comes directly from the API.

```
class App extends Component {

  ...

  render() {
    const { query, result } = this.state;
    return (
      <div className="page">
        <div className="interactions">
          <Search value={query} onChange={this.onSearchChange} onSubmit={this.on\
SearchSubmit}>
```

```
            Search
          </Search>
        </div>
        { result && <Table list={result.hits} /> }
      </div>
    );
  }
}

const Table = ({ list }) =>
  <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 behaviour on submit in a form. In React you will often come across the `preventDefault()` event function to suppress native browser behaviour.

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

Now you should be able to search different Hacker News stories. There should be no client-sided search anymore.

Exercises:

- read more about synthetic events in React[49]

[49]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[50] 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 query to the API.

Let's place a button below of the Table component to fetch more data. First you can define a very abstract and reusable Button component.

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

It might seem redundant to declare such a component. You have a `Button` instead of a `button` and only spare 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 behaviour for the button. Without the component you would have to refactor every button. Instead the Button component ensures to have one single source of truth. One Button to refactor all buttons at once.

Before you will use the Button, you have to declare more composable API constants to use the page parameter.

```
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.

```
`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${query}&${PARAM_PAGE}`

// https://hn.algolia.com/api/v1/search?query=redux&page=
```

The method `fetchSearchTopstories` will take the page as second argument. The `componentDid-Mount` and `onSearchSubmit` methods take the `DEFAULT_PAGE` for the initial and changed query API calls. They should fetch the first page whereas every fetch more should fetch the next page.

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

```
class App extends Component {

  ...

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

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

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

  ...

}
```

Now you can use the current page from the API response and the `fetchSearchTopstories` method in your 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 query and the current page + 1. The result will be the next page.

```
class App extends Component {

  ...

  render() {
    const { query, result } = this.state;
    const page = (result && result.page) || 0;
    return (
      <div className="page">
        <div className="interactions">
          <Search value={query} onChange={this.onSearchChange} onSubmit={this.on\
SearchSubmit}>
            Search
          </Search>
        </div>
        { result && <Table list={result.hits} /> }
        <div className="interactions">
          <Button onClick={() => this.fetchSearchTopstories(query, 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 the your old data. Instead of concatenating the old and new results, you replace it now. Let's adjust the functionality to add the new results 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 evaluate whether there are already old hits.

When the page is `0`, it's 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's the next page. The old hits are already stored in your state and thus can be used.

Third once you have the old hits, which you don't want to overwrite, you can merge them with the hits from the recent API request. The merge of both lists might look foreign to you. It uses the ES6 spread operator[51] to spread every value in both lists into a new list.

Last you set the merged hits and page in the internal component state. Make sure to understand what's happening here before you continue.

[51] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Spread_operator

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

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

- make sure to understand what you are doing in setSearchTopstories
- read more about the ES6 spread operator[52]
- experiment with the Hacker News API parameters[53]

[52]https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Spread_operator
[53]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. Since you already searched for "redux", you could spare the request and use a cached result from a previous request.

To implement that behaviour 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 query as key and result as value. Each result from the API will be saved by search query key.

Currently your result in the component state looks similar to the following:

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

Now you want to change it to:

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

Let's implement a client-side caching with React setState.

First rename the result object to results in the initial component state.

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      results: null,
      query: DEFAULT_QUERY,
    };

    ...

  }

  ...

}
```

Now you have to adjust the functionality where the result is stored to the internal component state. It should store multiple results.

```
class App extends Component {

  ...

  setSearchTopstories(result) {
    const { hits, page } = result;
    const { query } = this.state;

    const oldHits = page === 0 ? [] : this.state.results[query].hits;
    const updatedHits = [ ...oldHits, ...hits ];

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

  ...

}
```

In general the search query will be used as key to save the updated hits and page in a results map.

Therefore you have to retrieve the search query from the component state. 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 search query as key.

At the end a new result can be set in the results map in the internal component state. Let's examine the results object in setState.

```
results: { ...this.state.results, [query]: { hits: updatedHits, page } }
```

The right hand side makes sure to store the updated result by search query in the results map. The value is an object with a hits and page property. The key is the search query. The `[query]` notation might be new to you. It uses an ES6 dynamic key[54] to allocate values in an object.

The left hand side needs to spread all other results by search key in the state. Otherwise you would loose all results you stored before. That's what enables your cache.

Now you cache all results by search query.

In the next step you can retrieve the result depending on the search query from your map of results.

```
class App extends Component {

  ...

  render() {
    const { query, results } = this.state;
    const page = (results && results[query] && results[query].page) || 0;
    const list = (results && results[query] && results[query].hits) || [];
    return (
      <div className="page">
        <div className="interactions">
          <Search value={query} onChange={this.onSearchChange} onSubmit={this.on\
SearchSubmit}>
            Search
          </Search>
        </div>
        <Table list={list} />
        <div className="interactions">
          <Button onClick={() => this.fetchSearchTopstories(query, page + 1)}>
            More
          </Button>
        </div>
      </div>
```

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

```
    );
  }
}
```

Since you default to an empty list when there is no result by search query, you can spare the conditional check for the Table component.

The search functionality should work again. It caches all results from the Hacker News API. But something feels wrong when you test it on your own. The search component works a bit unexpected. On submit you make a server-side search request, but still the Table content adjusts by changing the search query in the input field without hitting the submit button.

It makes more sense to trigger search explicitly on submit without a client-side search. The issue is that the result depends on the search query. But the search query changes every time when you type something in the Search component. The query is a temporary property which changes all the time. On the other hand you want to have a fixed search key once you hit the submit button to keep a fixed search result. Let's introduce a search key.

```
class App extends Component {

  constructor(props) {
    super(props);

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

    ...
  }

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

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

  ...

}
```

The search key will be the fixed search query. It gets assigned before you make the Hacker News API request.

Now instead of the query property which is fluctuant, you will use the search key to get and set the result.

```
class App extends Component {

  ...

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

    const oldHits = page === 0 ? [] : this.state.results[searchKey].hits;
    const updatedHits = [ ...oldHits, ...hits ];

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

  ...

}
```

In the render lifecycle method you will use the search key as well. But make sure to keep the query property for the input field in the Search component which stays fluctuant.

```
class App extends Component {

  ...

  render() {
    const { query, 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={query} onChange={this.onSearchChange} onSubmit={this.on\
SearchSubmit}>
            Search
          </Search>
        </div>
        <Table list={list} />
        <div className="interactions">
          <Button onClick={() => this.fetchSearchTopstories(searchKey, page + 1)\
}>
            More
          </Button>
        </div>
      </div>
    );
  }
}
```

Now try it again. The search should only happen once you click the submit button. Additionally each result will be cached.

Still no one holds the API request back when you hit the submit button and there is already a cached result. What is a cached result when you don't use it? 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);
  }

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

  onSearchSubmit(event) {
    const { query } = this.state;
    this.setState({ searchKey: query });
    if (this.needsToSearchTopstories(query)) {
      this.fetchSearchTopstories(query, DEFAULT_PAGE);
    }
    event.preventDefault();
  }

  ...

}
```

Now your client hits the API only once although you search for a query twice. Even paginated data with several pages gets cached that way, because you always save the page for each result in the results map.

Exercises:

- read more about ES6 computed property names[55]

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

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,
      query: DEFAULT_QUERY,
      searchKey: '',
    };

    this.setSearchTopstories = this.setSearchTopstories.bind(this);
    this.fetchSearchTopstories = this.fetchSearchTopstories.bind(this);
    this.needsToSearchTopstories = this.needsToSearchTopstories.bind(this);
    this.onSearchChange = this.onSearchChange.bind(this);
    this.onSearchSubmit = this.onSearchSubmit.bind(this);
  }

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

    const oldHits = page === 0 ? [] : this.state.results[searchKey].hits;
    const updatedHits = [ ...oldHits, ...hits ];

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

```
    });
  }

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

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

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

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

  onSearchSubmit(event) {
    const { query } = this.state;
    this.setState({ searchKey: query });
    if (this.needsToSearchTopstories(query)) {
      this.fetchSearchTopstories(query, DEFAULT_PAGE);
    }
    event.preventDefault();
  }

  render() {
    const { query, 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={query} onChange={this.onSearchChange} onSubmit={this.on\
SearchSubmit}>
            Search
          </Search>
        </div>
        <Table list={list} />
        <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 }) =>
  <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: '15%' }}>
          {item.num_comments}
        </span>
        <span style={{ width: '15%' }}>
          {item.points}
        </span>
      </div>
    )}
  </div>
```

```
const Button = ({ onClick, children }) =>
  <button onClick={onClick} 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 rendering approaches
  - synthetic events on forms
  - reusable components like a Button
- 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 make 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 ready to implement your own higher order components and advanced interactions in React.

Snapshot Tests with Jest

Jest[56] is a JavaScript testing framework. At Facebook it's used to validate the JavaScript code. In the React community it's 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 you want to test during the chapter.

src/App.js

```
...

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 component renders without any errors.

src/App.test.js

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

it('renders without crashing', () => {
  const div = document.createElement('div');
  ReactDOM.render(<App />, div);
});
```

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

[56] 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.

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

```
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 render function.

Let's add more tests for our independent components.

```
...

import { Search, Button } 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();
  });

});

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();
  });

});
```

Moreover you can add tests for the Table.

```
...

import { 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 during the following chapters
- read more about Jest in React[57]

[57]https://facebook.github.io/jest/docs/tutorial-react.html

- read more about ES6 export[58] and import[59]

[58]https://developer.mozilla.org/en/docs/web/javascript/reference/statements/export
[59]https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Statements/import

Unit Tests with Enzyme

Enzyme[60] is a testing utility by Airbnb to assert, manipulate and traverse your React components. You can use it to conduct unit tests beside of 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 { shallow } from 'enzyme';

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);
  });

});
```

Shallow renders the component without children. 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`. They instantiate instances of the parent component and all child

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

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[61]

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

Loading ...

Now let's get back to your 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.

```
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,
      query: DEFAULT_QUERY,
      searchKey: '',
      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. Eventually the request will succeed and you can remove the loading state.

```
class App extends Component {

  ...

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

    const oldHits = page === 0 ? [] : this.state.results[searchKey].hits;
    const updatedHits = [ ...oldHits, ...hits ];

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

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

    fetch(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${query}&${PARAM_PAGE}${pag\
e}&${PARAM_HPP}${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 { query, 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={query} onChange={this.onSearchChange} onSubmit={this.on\
SearchSubmit}>
            Search
          </Search>
        </div>
        <Table list={result.hits} />
        <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 button to fetch more data appears. Once you fetch more data, the button will disappear and instead the Loading component will show up.

Exercises:

- use a library like Font Awesome[62] to show a loading icon instead of text

[62]http://fontawesome.io/

Higher Order Component

Higher order components (HOC) are equivalent to higher order functions. They take any input, mostly a component, and return a component as output. The component gets modified on the way.

Let's do a simple HOC which takes a component as input and returns a component.

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

In the example the input component would be 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, ...props }) =>
  isLoading ? <Loading /> : <Component { ...props } />;
```

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, ...props }` 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[63].

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 output is a ButtonWithLoading component.

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

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

const ButtonWithLoading = withLoading(Button);
```

As last step you have to use the ButtonWithLoading component which receives the loading state as additionally property. While the HOC consumes the loading property, the ‘onClick property gets passed to the Button component.

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

```
class App extends Component {

  ...

  render() {
    const { query, 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={query} onChange={this.onSearchChange} onSubmit={this.on\
SearchSubmit}>
            Search
          </Search>
        </div>
        <Table list={result.hits} />
        <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 enable multi-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
- implement another HOC

Advanced Sorting

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

It would be possible to write an own sort function, but personally I prefer to use utility libraries for such cases. Lodash[64] 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 { sortBy } from 'lodash';
```

You have four columns in your Table: title, author, comments and points. For each of them you can define a sort function 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(),
};
```

You can see that two of the sort functions are reverse. 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 to store 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.

[64] https://lodash.com/

```
this.state = {
  results: null,
  query: DEFAULT_QUERY,
  searchKey: '',
  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 which simply sets a sort key to your internal component state.

```
class App extends Component {

  constructor(props) {

    ...

    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.needsToSearchTopstories = this.needsToSearchTopstories.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 { query, 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={query} onChange={this.onSearchChange} onSubmit={this.on\
SearchSubmit}>
            Search
          </Search>
        </div>
        <Table list={list} sortKey={sortKey} onSort={this.onSort} />
        <div className="interactions">
          <ButtonWithLoading
            isLoading={isLoading}
            onClick={() => this.fetchSearchTopstories(searchKey, page + 1)}>
            More
          </ButtonWithLoading>
        </div>
      </div>
    );
  }
}
```

The Table component is responsible to sort 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 }) =>
  <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: '15%' }}>
          {item.num_comments}
        </span>
        <span style={{ width: '15%' }}>
          {item.points}
        </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 which uses Sort components to sort each column.

```
const Table = ({ list, sortKey, onSort }) =>
  <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: '15%' }}>
        <Sort sortKey={'COMMENTS'} onSort={onSort}>Comments</Sort>
      </span>
      <span style={{ width: '15%' }}>
        <Sort sortKey={'POINTS'} onSort={onSort}>Points</Sort>
      </span>
    </div>
    { SORTS[sortKey](list).map((item) =>
      ...
    )}
  </div>
```

Each Sort component gets a specific `sortKey` and the general `onSort` method. 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 by clicking on the column headers.

But a button as column header looks a bit stupid. Let's give the Sort a proper className.

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

Since a className doesn't get applied to a component, your abstract Button component needs to take care of it.

```
const Button = ({ onClick, className, children }) =>
  <button onClick={onClick} className={className} type="button">
    {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,
  query: DEFAULT_QUERY,
  searchKey: '',
  isLoading: false,
  sortKey: 'NONE',
  isSortReverse: false,
};
```

Now in your sort method you can evaluate if the list is reverse sorted. It's 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 { query, results, searchKey, isLoading, sortKey, isSortReverse } = thi\
s.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={query} onChange={this.onSearchChange} onSubmit={this.on\
SearchSubmit}>
            Search
          </Search>
        </div>
        <Table
          list={list}
          sortKey={sortKey}
          isSortReverse={isSortReverse}
          onSort={this.onSort}
        />
        <div className="interactions">
          <ButtonWithLoading
            isLoading={isLoading}
            onClick={() => this.fetchSearchTopstories(searchKey, page + 1)}>
            More
          </ButtonWithLoading>
        </div>
      </div>
    );
  }
}
```

The Table has to have a block body to compute the data now.

```
const Table = ({ list, sortKey, isSortReverse, onSort }) => {
  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 by now.

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

```
const Table = ({ list, sortKey, isSortReverse, onSort }) => {
  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}>Autho\
r</Sort>
        </span>
        <span style={{ width: '15%' }}>
          <Sort sortKey={'COMMENTS'} onSort={onSort} activeSortKey={sortKey}>Com\
ments</Sort>
```

```
        </span>
        <span style={{ width: '15%' }}>
          <Sort sortKey={'POINTS'} onSort={onSort} activeSortKey={sortKey}>Point\
s</Sort>
        </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 when it's 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(' ')} type\
="button">
      {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 classNames from 'classnames';
```

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} type="button">
      {children}
    </button>
  );
}
```

Your advanced sort interaction is complete now. Enjoy to sort the list!

Exercises:

- use a library like Font Awesome[65] to indicate the (reverse) sort
  - it could be an arrow up/down icon next to each Sort header
- read more about classnames library[66]
- implement another interaction

[65]http://fontawesome.io/
[66]https://github.com/JedWatson/classnames

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,
      query: DEFAULT_QUERY,
      searchKey: '',
      isLoading: false,
      sortKey: 'NONE',
      isSortReverse: false,
    };

    this.setSearchTopstories = this.setSearchTopstories.bind(this);
    this.fetchSearchTopstories = this.fetchSearchTopstories.bind(this);
    this.needsToSearchTopstories = this.needsToSearchTopstories.bind(this);
    this.onSearchChange = this.onSearchChange.bind(this);
    this.onSearchSubmit = this.onSearchSubmit.bind(this);
    this.onSort = this.onSort.bind(this);
  }

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

    const oldHits = page === 0 ? [] : this.state.results[searchKey].hits;
    const updatedHits = [ ...oldHits, ...hits ];

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

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

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

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

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

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

  onSearchSubmit(event) {
    const { query } = this.state;
    this.setState({ searchKey: query });
    if (this.needsToSearchTopstories(query)) {
      this.fetchSearchTopstories(query, DEFAULT_PAGE);
    }
    event.preventDefault();
  }

  onSort(sortKey) {
    const isSortReverse = this.state.sortKey === sortKey && !this.state.isSortRe\
verse;
    this.setState({ sortKey, isSortReverse });
  }

  render() {
    const { query, results, searchKey, isLoading, sortKey, isSortReverse } = thi\
s.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={query} onChange={this.onSearchChange} onSubmit={this.on\
SearchSubmit}>
            Search
          </Search>
        </div>
        <Table
          list={list}
          sortKey={sortKey}
          isSortReverse={isSortReverse}
          onSort={this.onSort}
        />
        <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>

const Table = ({ list, sortKey, isSortReverse, onSort }) => {
  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}>Autho\
r</Sort>
        </span>
        <span style={{ width: '15%' }}>
          <Sort sortKey={'COMMENTS'} onSort={onSort} activeSortKey={sortKey}>Com\
ments</Sort>
        </span>
        <span style={{ width: '15%' }}>
          <Sort sortKey={'POINTS'} onSort={onSort} activeSortKey={sortKey}>Point\
s</Sort>
        </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: '15%' }}>
            {item.num_comments}
          </span>
          <span style={{ width: '15%' }}>
            {item.points}
          </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} type="button">
      {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, ...props }) =>
  isLoading ? <Loading /> : <Component { ...props } />;

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 to write snapshot tests for your components
  - Enzyme allows to write unit tests for your components
  - higher order components are a common way to build advanced component
  - implementation of advanced interactions In React
  - conditional classNames with a neat helper library
- ES6
  - rest destructuring

Going Live

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 fulfil two requirements before you can deploy your app to Heroku:

- install the Heroku CLI[67]
- create a free Heroku account[68]

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[69]
- Heroku Buildpack for create-react-app[70]

[67]https://devcenter.heroku.com/articles/heroku-command-line
[68]https://www.heroku.com/
[69]https://blog.heroku.com/deploying-react-with-zero-configuration
[70]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! If I get positive feedback, I will continue to write on it.

But where can you continue? Before you dive into another book or tutorial, you should do your own hands on React project. Do it for one week, publish it somewhere and reach out to me on Twitter[71]. I am curious what you will build after you have read the book. You can also find me on GitHub[72] to share your repository.

After you have build your own React application, you may want to checkout Redux. Build your own SoundCloud Client in React + Redux[73]! In general I invite you to visit my website www.robinwieruch.de[74] to find more topics in web development.

If you liked the book, I hope to see you on social media to share it with you friends. I would really appreciate it.

[71]https://twitter.com/rwieruch
[72]https://github.com/rwieruch
[73]http://www.robinwieruch.de/the-soundcloud-client-in-react-redux
[74]http://www.robinwieruch.de/