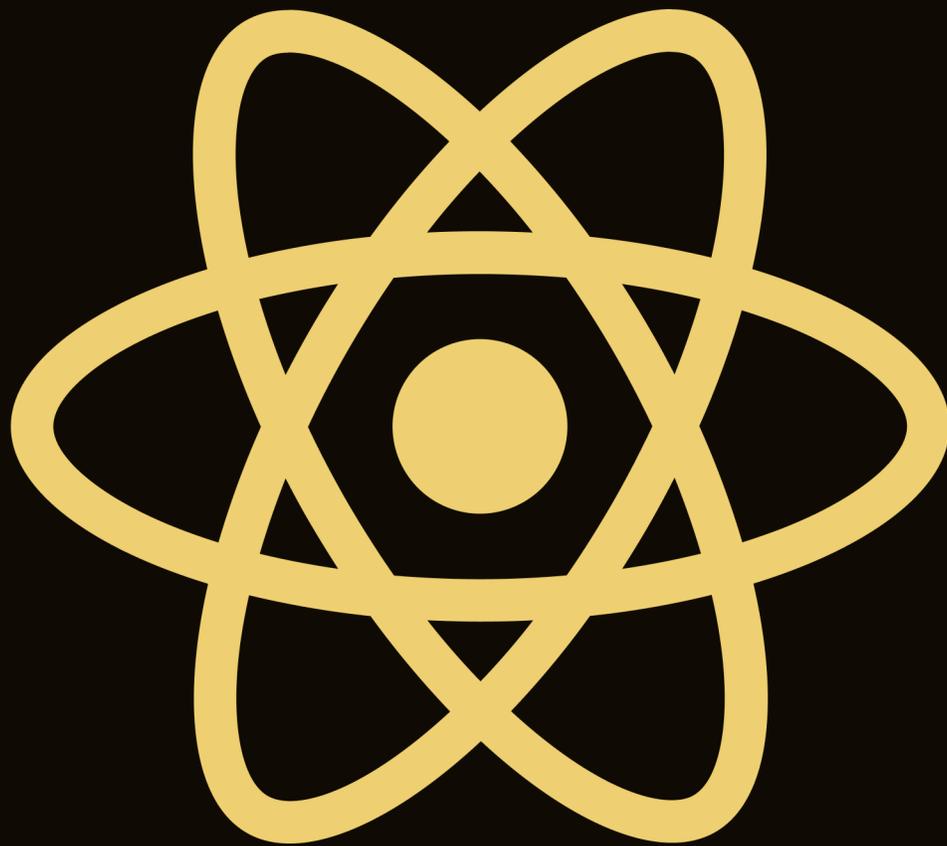


The Road to React



*2021 Edition
with React Hooks*

Required Knowledge: JavaScript

Robin Wieruch

The Road to React

Your journey to master React in JavaScript

Robin Wieruch

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

This version was published on 2021-10-20



Leanpub

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

© 2016 - 2021 Robin Wieruch

Tweet This Book!

Please help Robin Wieruch by spreading the word about this book on [Twitter!](#)

The suggested tweet for this book is:

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

The suggested hashtag for this book is [#ReactJs](#).

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

[#ReactJs](#)

Contents

Foreword	1
About the Author	2
FAQ	3
Who is this book for?	5
Fundamentals of React	6
Hello React	7
Requirements	9
Setting up a React Project	11
Meet the React Component	15
React JSX	19
Lists in React	22
Meet another React Component	28
React Component Instantiation	32
React DOM	35
React Component Definition (Advanced)	36
Handler Function in JSX	41
React Props	44
React State	47
Callback Handlers in JSX	50
Lifting State in React	52
React Controlled Components	57
Props Handling (Advanced)	60
React Side-Effects	71
React Custom Hooks (Advanced)	74
React Fragments	78
Reusable React Component	80
React Component Composition	83
Imperative React	86
Inline Handler in JSX	90
React Asynchronous Data	96
React Conditional Rendering	98
React Advanced State	102
React Impossible States	106

CONTENTS

Data Fetching with React	111
Data Re-Fetching in React	113
Memoized Handler in React (Advanced)	116
Explicit Data Fetching with React	118
Third-Party Libraries in React	121
Async/Await in React (Advanced)	123
Forms in React	125
React's Legacy	129
React Class Components	130
React Class Components: State	133
Imperative React	135
Styling in React	137
CSS in React	139
CSS Modules in React	145
Styled Components in React	151
SVGs in React	157
React Maintenance	160
Performance in React (Advanced)	161
TypeScript in React	170
Testing in React	180
React Project Structure	207
Real World React (Advanced)	212
Sorting	213
Reverse Sort	219
Remember Last Searches	222
Paginated Fetch	232
Deploying a React Application	242
Build Process	243
Deploy to Firebase	244
Outline	247

Foreword

The Road to React teaches the fundamentals of React. You will build a real-world application in plain React without complicated tooling. Everything from project setup to deployment on a server will be explained to you. The book comes with additional referenced reading material and exercises with each chapter. After reading the book, you will be able to build your own applications in React. The material is kept up to date by myself and the community. In the Road to React, I offer a foundation before you dive into the broader React ecosystem. It explains general concepts, patterns, and best practices for a real-world React application. Essentially, you will learn to build your own React application from scratch, with features like pagination, client-side and server-side searching, and advanced interactions like sorting. I hope this book captures my enthusiasm for React and JavaScript, and that it helps you get started with it.

About the Author

I am a German software and web developer dedicated to learning and teaching JavaScript. After obtaining my Master's Degree in computer science, I gained experience from the startup world where I used JavaScript intensively during both my professional life and spare time. For a few years, I worked closely with an exceptional team of engineers at a company in Berlin, developing large-scale JavaScript applications, which eventually led to a desire to teach others about these topics.

During my time as a developer in Berlin, I regularly wrote articles about web development on my website. I received great feedback from people learning from my articles which allowed me to improve my writing and teaching style. Article after article, I grew my ability to teach others. I find it it fulfilling to see students strive by giving them clear objectives and short feedback loops.

Currently, I am a self-employed developer helping companies in creating their MVPs, conducting workshops and code audits/reviews, and creating large-scale JavaScript applications. You can find more information about me, ways to support me, and how to work with me on my [website](https://www.robinwieruch.de/about)¹.

¹<https://www.robinwieruch.de/about>

FAQ

How to get updates?

I have two channels where I share updates about my content. You can [subscribe to updates by email](#)² or [follow me on Twitter](#)³. Regardless of the channel, my objective is to only share quality content. Once you receive a notification about an update, you can download a new version of the book from my website.

Is the learning material up-to-date?

Programming books are usually outdated soon after their release, but since this book is self-published, I can update it as needed whenever a new version of something related to this book gets released.

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

If you have bought the book on Amazon, you may have seen that the book is available on my website too. Since I use Amazon as one way to monetize my often free content, I honestly thank you for your support and invite you to sign up for my courses on [my website](#)⁴. After creating an account there, write me an email about your purchase with a receipt from Amazon, so that I can unlock the content for you. With an account on my platform, you always have access to the latest version of the book.

Why is the print copy so large in size?

If you have purchased the print version of the book, make sure to take notes in the book. It was my intention to keep the printed book extra-large, for the sake of giving larger code snippets enough space, but also for giving you enough space to work with it.

How can I get help while reading the book?

The book has a community of learners who help each other and for people who are reading along. You can join the community to get help or to help others, as helping others may help you internalize your own understanding. Just follow the navigation to my courses on my [website](#)⁵, sign up there, and navigate to joining the community.

Can I help to improve the content?

If you have feedback, shoot me an email and I will consider your suggestions. Don't expect many replies for bug tracking or troubleshoots though because that's what's the community for.

What do I do if I encounter a bug?

If you encounter any bug in the code, you should find a URL to the current GitHub project at the end of each section. Feel free to open a GitHub issue there. Your help is very much appreciated!

²<https://www.getrevue.co/profile/rwieruch>

³<https://twitter.com/rwieruch>

⁴<https://www.robinwieruch.de>

⁵<https://www.robinwieruch.de>

How to support the project?

If you find my lessons useful and would like to contribute, seek my website's [about page](#)⁶ for information about how to offer support. It is also very helpful for my readers to spread the word about how my books helped them, so others might discover it as ways to improve their web development skills. Contributing through any of the provided channels gives me the freedom to create in-depth courses, and to continue offering free material on my website.

What's your motivation behind the book?

I want to teach about this topic consistently. I often find materials online that don't receive updates, or only apply to a small part of a topic. Sometimes people struggle to find consistent and up-to-date resources to learn from. I want to provide this consistent and up-to-date learning experience. Also, I hope I can support the less fortunate with my projects by giving them the content for free or by [having other impacts](#)⁷.

⁶<https://www.robinwieruch.de/about/>

⁷<https://www.robinwieruch.de/giving-back-by-learning-react>

Who is this book for?

JavaScript Beginners

JavaScript beginners with knowledge in fundamental JS, CSS, and HTML: If you just started out with web development, and have a basic grasp of JS, CSS, and HTML, this book should give you everything that's needed to learn React. However, if you feel there is a gap in your JavaScript knowledge, don't hesitate to read up on that topic before continuing with the book. You will have lots of references to fundamental JavaScript knowledge in this book though.

JavaScript Veterans

JavaScript veterans coming from jQuery: If you have used JavaScript with jQuery, MooTools, and Dojo extensively back in the days, the new JavaScript era may seem overwhelming for someone getting back on track with it. However, most of the fundamental knowledge didn't change, it's still JavaScript and HTML under the hood, so this book should give you the right start into React.

JavaScript Enthusiasts

JavaScript enthusiasts with knowledge in other modern SPA frameworks: If you are coming from Angular or Vue, there may be lots of differences in how to write applications with React, however, all these frameworks share the same fundamentals of JavaScript and HTML. After a mindset shift to get comfortable with React, you should be doing just fine adopting React.

Non-JavaScript Developers

If you are coming from another programming language, you should be more familiar than others with the different aspects of programming. After picking up the fundamentals of JavaScript and HTML, you should have a good time learning React with me.

Designers and UI/UX Enthusiasts

If your main profession is in design, user interaction, or user experience, don't hesitate to pick up this book. You may be already quite familiar with HTML and CSS which is a plus. After going through some more JavaScript fundamentals, you should be good to get through this book. These days UI/UX is moving closer to the implementation details which are often taken care of with React. It would be your perfect asset to know how things work in code.

Team Leads, Product Owners, or Product Managers

If you are a team lead, product owner or product manager of your development department, this book should give you a good breakdown of all the essential parts of a React application. Every section explains one React concept/pattern/technique to add another feature or to improve the overall architecture. It's a well-rounded reference guide for React.

Fundamentals of React

In this first part of this learning experience, we'll cover the fundamentals of React, with which we'll create our first React project. Later we'll explore new use cases for React by implementing real features like client and server-side searching, remote data fetching, and advanced state management the same as developing an actual web application. By the end, you will have a fully running React application that interacts with real-world data.

Hello React

Single-page applications (SPA⁸) have become increasingly popular with first-generation SPA frameworks like Angular (by Google), Ember, Knockout, and Backbone. Using these frameworks made it easier to build web applications that advanced beyond vanilla JavaScript and jQuery. React, yet another solution for SPAs, was released by Facebook later in 2013. All of them are used to create web applications in JavaScript.

For a moment, let's go back in time before SPAs existed: In the past, websites and web applications were rendered from the server. A user visits a URL in a browser and requests one HTML file and all its associated HTML, CSS, and JavaScript files from a web server. After some network delay, the user sees the rendered HTML in the browser (client) and starts to interact with it. Every additional page transition (meaning: visiting another URL) would initiate this chain of events again. In this version from the past, essentially everything crucial is done by the server, whereas the client plays a minimal role by just rendering page by page. While barebones HTML and CSS were used to structure the application, just a little bit of JavaScript was thrown into the mix to make interactions (e.g. toggling a dropdown) or advanced styling (e.g. positioning a tooltip) possible. A popular JavaScript library for this kind of work was jQuery.

In contrast, modern JavaScript shifted the focus from the server to the client. The most extreme version of it: A user visits a URL and requests *one small HTML file* and *one larger JavaScript file*. After some network delay, the user sees the *rendered HTML by JavaScript* in the browser and starts to interact with it. Every additional page transition *wouldn't* request more files from the web server, but would use the initially requested JavaScript to render the new page. Also, every additional interaction by the user is handled on the client too. In this modern version, the server delivers mainly JavaScript across the wire with one minimal HTML file. The HTML file then executes all the linked JavaScript on the client-side to render the entire application with HTML and JavaScript for its interactions.

React, among the other SPA solutions, makes this possible. Essentially a SPA is one bulk of JavaScript, which is neatly organized in folders and files, to create a whole application whereas the SPA framework (e.g. React) gives you all the tools to architect it. This JavaScript-focused application is delivered once over the network to your browser when a user visits the URL for your web application. From there, React, or any other SPA framework, takes over for rendering everything in the browser and for dealing with user interactions.

With the rise of React, the concept of components became popular. Every component defines its look and feel with HTML, CSS, and JavaScript. Once a component is defined, it can be used in a hierarchy of components for creating an entire application. Even though React has a strong focus on components as a library, the surrounding ecosystem makes it a flexible framework. React has a slim API, a stable yet thriving ecosystem, and a welcoming community. We are happy to welcome you :-)

⁸<https://bit.ly/3BZOL1o>

Exercises

- Read more about [how we moved from websites to web applications](#)⁹.
- Read more about [why I moved from Angular to React](#)¹⁰.
- Read more about [how to learn a framework](#)¹¹.
- Read more about [how to learn React](#)¹².
- Read more about [JavaScript fundamentals needed for React](#)¹³ – without worrying too much about React here – to challenge yourself with several JavaScript features used in React.
- Optional: [Rate this section](#)¹⁴.

⁹<https://www.robinwieruch.de/web-applications>

¹⁰<https://www.robinwieruch.de/reasons-why-i-moved-from-angular-to-react>

¹¹<https://www.robinwieruch.de/how-to-learn-framework>

¹²<https://www.robinwieruch.de/learn-react-js>

¹³<https://www.robinwieruch.de/javascript-fundamentals-react-requirements>

¹⁴<https://forms.gle/NTqhvYDaP1RjtanC6>

Requirements

To follow this book, you'll need to be familiar with the basics of web development, i.e how to use HTML, CSS, and JavaScript. It also helps to understand [APIs](#)¹⁵, as they will be covered in this learning experience. Along with these skills, you'll need the following tools to code with me while reading this book.

Editor and Terminal

I have provided a [setup guide](#)¹⁶ to get you started with general web development. For this learning experience, you will need a text editor (e.g. Sublime Text) and a command line tool (e.g. iTerm). As an alternative, I recommend using an IDE, for example **Visual Studio Code** (also called VSCode), for beginners, as it's an all-in-one solution that provides an advanced editor with an integrated command line tool, and because it's a popular choice among web developers.

If you don't want to set up the editor/terminal combination or IDE on your local machine, [CodeSandbox](#)¹⁷, an online editor, is also a viable alternative. While CodeSandbox is a great tool for sharing code online, a local machine setup is a better tool for learning the different ways to create a web application. Also, if you want to develop applications professionally, a local setup will be required.

Throughout this learning experience, I will use the term *command line*, which will be used synonymously for the terms *command line tool*, *terminal*, and *integrated terminal*. The same applies to the terms *editor*, *text editor*, and *IDE*, depending on what you decided to use for your setup.

Optionally, I recommend managing projects on **GitHub** while we conduct the exercises in this book, and I've provided a [short guide](#)¹⁸ on how to use these tools. Github has excellent version control, so you can see what changes were made if you make a mistake or just want a more direct way to follow along. It's also a great way to share your code later with other people.

Node and NPM

Before we can begin, we'll need to have **Node and NPM**¹⁹ installed. Both are used to manage libraries (node packages) that you will need along the way. These node packages can be libraries or whole frameworks. We'll install external node packages via npm (node package manager).

You can verify node and npm versions in the command line using the `node --version` and `npm --version` commands. If you don't receive output in the terminal indicating which version is installed, you need to install node and npm:

¹⁵<https://www.robinwieruch.de/what-is-an-api-javascript>

¹⁶<https://www.robinwieruch.de/developer-setup>

¹⁷<https://codesandbox.io>

¹⁸<https://www.robinwieruch.de/git-essential-commands>

¹⁹<https://nodejs.org/en/>

Command Line

```
node --version
*vXX.YY.ZZ
npm --version
*vXX.YY.ZZ
```

If you have already installed Node and npm, make sure that your installation is the most recent version. If you're new to npm or need a refresher, this [npm crash course](https://www.robinwieruch.de/npm-crash-course)²⁰ I created will get you up to speed.

²⁰<https://www.robinwieruch.de/npm-crash-course>

Setting up a React Project

In the Road to React, we'll use [create-react-app](https://bit.ly/3jlkzHd)²¹ to bootstrap your application. It's an opinionated yet zero-configuration starter kit for React introduced by Facebook in 2016, which is [recommended for beginners by 96% of React users](https://bit.ly/3AY58u3)²². In *create-react-app*, the tooling and configuration evolve in the background, while the focus remains on the application's implementation.

After installing Node and NPM, use the command line to type the following command in a dedicated folder for your project. We'll refer to this project as *hacker-stories*, but you may choose any name you like:

Command Line

```
npx create-react-app hacker-stories
```

Navigate into your new folder after the setup has finished:

Command Line

```
cd hacker-stories
```

Now we can open the application in an editor or IDE. For Visual Studio Code, you can simply type `code .` on the command line. The following folder structure, or a variation of it depending on the *create-react-app* version, should be presented:

Project Structure

```
hacker-stories/  
--node_modules/  
--public/  
--src/  
----App.css  
----App.js  
----App.test.js  
----index.css  
----index.js  
----logo.svg  
--.gitignore  
--package-lock.json  
--package.json  
--README.md
```

This is a breakdown of the most important folders and files:

²¹<https://bit.ly/3jlkzHd>

²²<https://bit.ly/3AY58u3>

- **README.md:** The *.md* extension indicates the file is a markdown file. Markdown is a lightweight markup language with plain text formatting syntax. Many projects come with a *README.md* file that gives instructions and useful information about the project. When we push projects to platforms like GitHub, the *README.md* file usually displays information about the content contained in its repositories. Because you used create-react-app, your *README.md* should be the same as the official's [create-react-app GitHub repository](https://github.com/facebook/create-react-app)²³.
- **node_modules/:** This folder contains all node packages that have been installed. Since we used create-react-app, a couple of node modules are already installed. We'll not touch this folder, since node packages are usually installed and uninstalled with npm via the command line.
- **package.json:** This file shows you a list of node package dependencies and other project configurations.
- **package-lock.json:** This file indicates npm how to break down all node package versions. We'll not touch this file.
- **.gitignore:** This file displays all files and folders that shouldn't be added to your git repository when using git, as such files and folders should be located only in your local project. The *node_modules/* folder is one example. It is enough to share the *package.json* file with others, so they can install dependencies on their end with `npm install` without your entire dependency folder.
- **public/:** This folder holds development files, such as *public/index.html*. The index file is displayed on *localhost:3000* when the app is in development or on a domain that is hosted elsewhere. The default setup handles relating this *index.html* with all the JavaScript from *src/*.

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

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

After you have learned about the folder and file structure of your React project, let's go through the available commands to get it started. All your project-specific commands can be found in your *package.json* under the `scripts` property. They may look similar to these:

²³<https://bit.ly/3jjkzHd>

package.json

```
{
  ...
},
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject"
},
  ...
}
```

These scripts are executed with the `npm run <script>` command in an IDE-integrated terminal or your standalone command line tool. The `run` can be omitted for the `start` and `test` scripts. The commands are as follows:

Command Line

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

```
npm start
```

```
# Runs the tests
```

```
npm test
```

```
# Builds the application for production
```

```
npm run build
```

Another command from the previous `npm` scripts called `eject` shouldn't be used for this learning experience. It's a one-way operation, because once you `eject`, you can't go back. Essentially this command is only there to make all the tooling and configuration from `create-react-app` accessible if you are not satisfied with the choices or if you want to change something. Here we will keep all the defaults though.

Exercises:

- Read a bit more through React's [create-react-app documentation](https://bit.ly/3jjkzHd)²⁴ and [getting started guide](https://create-react-app.dev/docs/getting-started)²⁵.
 - Read more about [the supported JavaScript features in create-react-app](https://bit.ly/3vvl4Tn)²⁶.
- Read more about [the folder structure in create-react-app](https://bit.ly/3jeBN8H)²⁷.

²⁴<https://bit.ly/3jjkzHd>

²⁵<https://create-react-app.dev/docs/getting-started>

²⁶<https://bit.ly/3vvl4Tn>

²⁷<https://bit.ly/3jeBN8H>

- Go through all of your React project’s folders and files one by one.
- Read more about [the scripts in create-react-app](#)²⁸.
 - Start your React application with `npm start` on the command line and check it out in the browser.
 - * Exit the command on the command line by pressing `Control + C`.
 - Run the `npm test` script.
 - Run the `npm run build` script and verify that a *build/* folder was added to your project (you can remove it afterward). Note that the build folder can be used later on to [deploy your application](#)²⁹.
- Every time we change something in our code throughout the coming learning experience, make sure to check the output in your browser for getting visual feedback. Use `npm start` to keep your application running.
- Optionally: If you use git and GitHub, add and commit your changes after every section of the book.
- Optional: [Rate this section](#)³⁰.

²⁸<https://bit.ly/3vvjsJx>

²⁹<https://www.robinwieruch.de/deploy-applications-digital-ocean/>

³⁰<https://forms.gle/bvH2jcppsSA6p9i16>

Meet the React Component

Every React application is built on the foundation of React components. In this section, you will get to know your first React component which is located in the `src/App.js` file and which should look similar to the example below. Depending on your create-react-app version, the content of the file might differ slightly:

`src/App.js`

```
import * as React from 'react';
import logo from './logo.svg';
import './App.css';

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

export default App;
```

This file will be our focus throughout this tutorial, unless otherwise specified. Let's start by reducing the component to a more lightweight version for getting you started without too much [boilerplate code](#)³¹. Afterward, start your application with `npm start` in the command line and check what's displayed in the browser:

³¹<https://bit.ly/3lZzckS>

src/App.js

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

function App() {
  return (
    <div>
      <h1>Hello World</h1>
    </div>
  );
}

export default App;
```

Before we dive deeper into each topic, here comes a quick overview of what you are seeing:

- First, this React component, called the App component, is just a JavaScript function. In contrast to JavaScript functions, it's defined in [PascalCase](#)³². This kind of component is commonly called a **function component**. Function components are the modern way of using components in React, however, be aware that there are other variations of React components too. (see **component types** later)
- Second, the App component doesn't receive any parameters in its function signature yet. In the upcoming sections, you will learn how to pass information (props) from one component to another component. These props will be accessible via the function's signature as parameters then. (see **props** later)
- And third, the App component returns code that resembles HTML. You will see how this new syntax, called JSX, allows you to combine JavaScript and HTML for displaying highly dynamic and interactive content in a browser. (see **JSX** later)

The function component possesses implementation details between the function signature and the return statement, like any other JavaScript function. You will see this in practice in action throughout your React journey:

³²<https://www.robinwieruch.de/javascript-naming-conventions>

src/App.js

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

function App() {
  // you can do something in between

  return (
    <div>
      <h1>Hello World</h1>
    </div>
  );
}

export default App;
```

Variables defined in the function's body will be re-defined each time this function runs, which shouldn't be something new if you are familiar with JavaScript and its functions:

src/App.js

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

function App() {
  const title = 'React';

  return (
    <div>
      <h1>Hello World</h1>
    </div>
  );
}

export default App;
```

The function of a function component will either run when a component is rendered for the first time or re-rendered when it updates. The act of rendering means that the component displays itself in the browser. Since we don't need anything from within the App component that will be used to define this variable from the last code snippet – for example parameters coming from the component's function signature – we can define the variable outside of the App component as well. Thus it will be defined only once and not every time the function is called:

src/App.js

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

const title = 'React';

function App() {
  return (
    <div>
      <h1>Hello World</h1>
    </div>
  );
}

export default App;
```

On your journey as a React developer, and in this learning experience, you will come across both scenarios: variables defined outside and within a component. As a rule of thumb: If a variable does not need anything from within the function component's body (e.g. parameters), then define it outside of the component which avoids redefining it on every function call.

Exercises:

- Confirm your [source code](#)³³.
- If you are unsure when to use `const`, `let` or `var` in JavaScript (or React) for variable declarations, make sure to [read more about their differences](#)³⁴.
 - Read more about `const`³⁵.
 - Read more about `let`³⁶.
- Think about ways to display the `title` variable in your `App` component's returned HTML. In the next section, we'll put this variable to use.
- Optional: [Rate this section](#)³⁷.

³³<https://bit.ly/3G6O6gX>

³⁴<https://www.robinwieruch.de/const-let-var>

³⁵<https://mzl.la/3jkvHn5>

³⁶<https://mzl.la/3piHQgc>

³⁷<https://forms.gle/VYiZqqjzXGE11wCv6>

React JSX

Everything returned from a React component will be displayed in the browser. Until now, we only returned HTML from the App component. However, recall that I mentioned the returned output of the App component not only resembles HTML, but it can also be mixed with JavaScript. In fact, this output is called JSX (JavaScript XML), powerfully combines HTML and JavaScript. Let's see how this works for displaying the variable from the previous section:

src/App.js

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

const title = 'React';

function App() {
  return (
    <div>
      <h1>Hello {title}</h1>
    </div>
  );
}

export default App;
```

Either start your application again with `npm start` (or check whether your application still runs) and look for the rendered variable in the browser, which should read: "Hello React". If you change the variable in the source code, the browser should reflect that change.

Now let's focus on the HTML which differs slightly in JSX. An HTML input field with a label can be defined as follows:

src/App.js

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

const title = 'React';

function App() {
  return (
    <div>
      <h1>Hello {title}</h1>

      <label htmlFor="search">Search: </label>
      <input id="search" type="text" />
    </div>
  );
}
```

```
        </div>
      );
    }

export default App;
```

For our input field and label combination, we specified three HTML attributes: `htmlFor`, `id`, and `type`. Where `id` and `type` should be familiar from native HTML, `htmlFor` might be new to you. The `htmlFor` reflects the `for` attribute in HTML. JSX replaces a handful of internal HTML attributes caused by internal implementation details of React itself. However, you can find all the [supported HTML attributes](#)³⁸ in React's documentation, which follows the [camel case](#)³⁹ naming convention. Expect to come across more JSX-specific attributes like `className` and `onClick` instead of `class` and `onclick`, as you learn more about React.

We will revisit the HTML input field for implementation details later; for now, let's return to JavaScript in JSX in contrast to HTML. We have defined a JavaScript string primitive to be displayed in the `App` component, and the same can be done with a JavaScript object:

src/App.js

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

const welcome = {
  greeting: 'Hey',
  title: 'React',
};

function App() {
  return (
    <div>
      <h1>
        {welcome.greeting} {welcome.title}
      </h1>

      <label htmlFor="search">Search: </label>
      <input id="search" type="text" />
    </div>
  );
}

export default App;
```

³⁸<https://bit.ly/2Z42zcK>

³⁹<https://bit.ly/3jljQFn>

Essentially everything in curly braces in JSX can be used for JavaScript. For example, executing a function works this way too:

`src/App.js`

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

function getTitle(title) {
  return title;
}

function App() {
  return (
    <div>
      <h1>Hello {getTitle('React')}</h1>

      <label htmlFor="search">Search: </label>
      <input id="search" type="text" />
    </div>
  );
}

export default App;
```

JSX was initially invented for React, but it became useful for other modern libraries and frameworks after it gained popularity. [It's one of my favorite things about React⁴⁰](#). Without any extra templating syntax (except for the curly braces), we are now able to use JavaScript in HTML. Every data structure, from JavaScript primitive to a JavaScript function, can be displayed within HTML with the help of JSX.

Exercises:

- Confirm your [source code⁴¹](#).
 - Confirm the [changes⁴²](#).
- Read more about [React's JSX⁴³](#).
- Read more about [JavaScript Variables⁴⁴](#).
 - Define more primitive and complex JavaScript data types and render them in JSX.
 - Try to render a JavaScript array in JSX ([hint⁴⁵](#)). If it's too complicated, don't worry, because you will learn more about this in the next section.
- Optional: [Rate this section⁴⁶](#).

⁴⁰<https://bit.ly/3aZbdM0>

⁴¹<https://bit.ly/3vvS8ec>

⁴²<https://bit.ly/3n3WW6o>

⁴³<https://bit.ly/3BZSkVk>

⁴⁴<https://www.robinwieruch.de/javascript-variable>

⁴⁵<https://mzl.la/3B3a7tf>

⁴⁶<https://forms.gle/R6y6kEqGPACLRXmP8>

Lists in React

So far we've rendered a few variables in JSX, however, most often you will deal with arrays. Thus we'll render a list of items next. We'll experiment with sample data first, and later we'll apply that knowledge to fetched data from a remote API.

First, let's define the array as a variable. Similar as before, we can define a variable outside or inside the component. The following defines it outside:

```
src/App.js
```

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

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

function App() { ... }

export default App;
```

Note: I used a ... here as a placeholder, to keep my code snippet small (without the App component's implementation details) and focused on the essential parts (the list variable outside of the App component). I will use the ... throughout the rest of this learning experience as a placeholder for code blocks that I have established in previous sections. If you get lost, you can always verify your code using the CodeSandbox links I provide at the end of most sections.

Each item in the list has a title, a url, an author, an identifier (objectID), points – which indicate the popularity of an item – and a count of comments (num_comments). Next, we'll render the list within our JSX dynamically:

src/App.js

```
function App() {
  return (
    <div>
      <h1>My Hacker Stories</h1>

      <label htmlFor="search">Search: </label>
      <input id="search" type="text" />

      <hr />

      { /* render the list here */ }
      { /* and by the way: that's how you do comments in JSX */ }
    </div>
  );
}
```

You can use the [built-in JavaScript map method for arrays](#)⁴⁷ to iterate over each item of the list and return a new version of each:

Code Playground

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

const newNumbers = numbers.map(function(number) {
  return number * 2;
});

console.log(newNumbers);
// [2, 8, 18, 32]
```

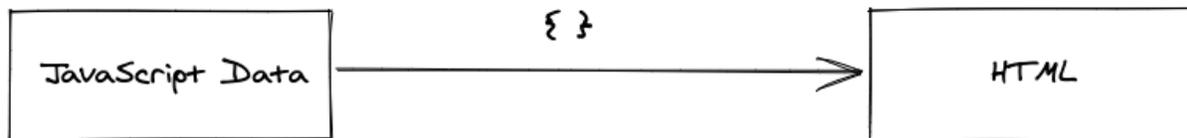
In our case, we won't map from one JavaScript data type to another. Instead, we return a JSX fragment that renders each item of the list:

⁴⁷<https://mzl.la/3B3a7tf>

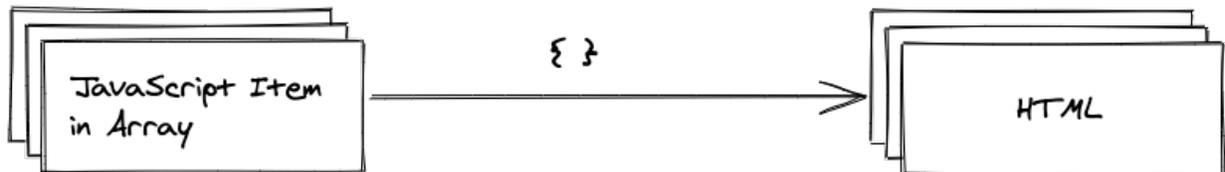
src/App.js

```
function App() {  
  return (  
    <div>  
      ...  
  
    <hr />  
  
    <ul>  
      {list.map(function (item) {  
        return <li>{item.title}</li>;  
      })}  
    </ul>  
  </div>  
);  
}
```

Actually, one of my first React “Aha” moments was using barebones JavaScript to map a list of JavaScript objects to HTML elements without any other HTML templating syntax. It’s just JavaScript mixed with HTML.



Example:



React will display each item now, but you can still improve your code so React handles dynamic lists more gracefully. By assigning a key attribute to each list item’s element, React can identify items if the list changes (e.g. re-ordering). The key isn’t necessary yet in our current situation, however, it’s a best practice to use it from the start. Fortunately, our items come with an identifier:

src/App.js

```
function App() {
  return (
    <div>
      ...

      <hr />

      <ul>
        {list.map(function (item) {
          return <li key={item.objectID}>{item.title}</li>;
        })}
      </ul>
    </div>
  );
}
```

We avoid using the `index` of the item in the array to make sure the key attribute is a stable identifier. If the list changes its order, for example, React will not be able to identify the items properly when using the array's `index`:

Code Playground

```
// avoid doing this
<ul>
  {list.map(function (item, index) {
    return (
      <li key={index}>
        ...
      </li>
    );
  })}
</ul>
```

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

src/App.js

```
function App() {
  return (
    <div>
      ...

      <hr />

      <ul>
        {list.map(function (item) {
          return (
            <li 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>
            </li>
          );
        })}
      </ul>
    </div>
  );
}
```

The map function is inlined concisely in your JSX. Within the map function, we have access to each item and its properties. The `url` property of each item is used as dynamic `href` attribute for the HTML anchor tag. Not only can JavaScript in JSX be used to display items, but also to assign HTML attributes dynamically. This section only scratches the surface of how powerful it is to use mixed JavaScript and HTML, however, using an array's map function or assigning dynamic HTML attributes should give you a good first impression.

Exercises:

- Confirm your [source code](#)⁴⁸.
 - Confirm the [changes](#)⁴⁹.
- Read more about why React's key attribute is needed ([0](#)⁵⁰, [1](#)⁵¹). Don't worry if you don't understand the implementation yet, just focus on what problem it causes for dynamic lists.

⁴⁸<https://bit.ly/2Z6e2ZI>

⁴⁹<https://bit.ly/3jf7a2Q>

⁵⁰<https://www.robinwieruch.de/react-list-key>

⁵¹<https://bit.ly/3vsA1pj>

- Recap the [standard built-in array methods](#)⁵² – especially *map*, *filter*, and *reduce* – which are available in native JavaScript.
- What happens if you return `null` instead of the JSX? Try it first, then read the answer:
 - Returning `null` in JSX is allowed. It's always used if you want to render nothing.
- Extend the list with some more items to make the example more realistic.
- Practice using different JavaScript expressions in JSX.
- Optional: [Rate this section](#)⁵³.

⁵²<https://mzl.la/3b9V9rf>

⁵³<https://forms.gle/aZmLFjEdSMTk9Thk9>

Meet another React Component

Components are the foundation of every React application. So far we've only been using the App component. This will not end up well, because components should scale with your application's size. So instead of making one component larger and more complex, we'll split one component into multiple components eventually. We'll start with a new List component which extracts functionalities from the App component:

src/App.js

```
const list = [ ... ];

function App() { ... }

function List() {
  return (
    <ul>
      {list.map(function (item) {
        return (
          <li 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>
          </li>
        );
      })}
    </ul>
  );
}
```

Then the new List component can be used in the App component where we have been using the inlined functionality previously:

src/App.js

```
function App() {
  return (
    <div>
      <h1>My Hacker Stories</h1>

      <label htmlFor="search">Search: </label>
      <input id="search" type="text" />

      <hr />

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

You've just created your first React component! With this example in mind, we can see how components encapsulate meaningful tasks while contributing to the greater good of a larger React application. Extracting a component is a task that you will perform very often as a React developer because it's always the case that a component will grow in size and complexity. Let's do this extraction of a component one more time for a so-called Search component:

src/App.js

```
function App() {
  return (
    <div>
      <h1>My Hacker Stories</h1>

      <Search />

      <hr />

      <List />
    </div>
  );
}

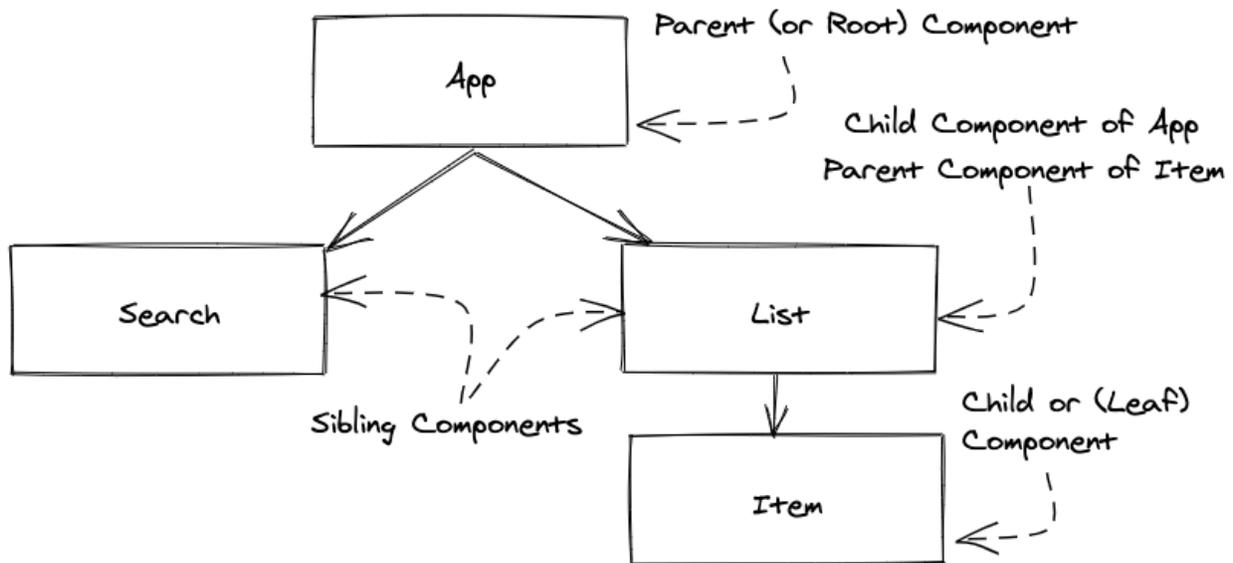
function Search() {
  return (
    <div>
      <label htmlFor="search">Search: </label>
      <input id="search" type="text" />
    </div>
  );
}
```

```

    </div>
  );
}

```

Finally, we have three components in our application: App, List, and Search. Generally speaking, a React application consists of many hierarchical components; which we can put into the following categories:



React applications have **component hierarchies** (also called **component trees**). There is usually one uppermost **entry point component** (e.g. App) that spans a tree of components below it. The App is the **parent component** of the List and Search, so the List and Search are **child components** of the App component and **sibling components** to each other. The illustration takes it one step further where the Item component is a child component of the List. In a component tree, there is always a **root component** (e.g. App), and the components that don't render any other components are called **leaf components** (e.g. List/Search component in our current source code or Item/Search component from the illustration). All components can have zero, one, or many child components.

Exercises:

- Confirm your [source code](https://bit.ly/3plb66a)⁵⁴.
 - Confirm the [changes](https://bit.ly/3G61kKU)⁵⁵.
- Ask yourself:
 - What problem could arise if we keep treating the `list` variable as a global variable. Think about a way how to prevent it.

⁵⁴<https://bit.ly/3plb66a>

⁵⁵<https://bit.ly/3G61kKU>

- We can't extract an Item component from the List component (like in the illustration) yet, because we don't know how to pass individual items from the list to each Item component. Think about a way to do it.
- Optional: [Rate this section](#)⁵⁶.

⁵⁶<https://forms.gle/EZENmy48zvDP82NL7>

React Component Instantiation

Next, I'll briefly explain JavaScript classes, to help clarify React components. Technically they are not related, which is important to note, but it is a fitting analogy for you to understand the concept of a component.

Classes are most often used in object-oriented programming languages. JavaScript, always flexible in its programming paradigms, allows functional programming and object-oriented programming to co-exist side-by-side. To recap JavaScript classes for object-oriented programming, consider the following *Developer* class:

Code Playground

```
class Developer {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  getName() {
    return this.firstName + ' ' + this.lastName;
  }
}
```

Each class has a constructor that takes arguments and assigns them to the class instance. A class can also define functions that are associated with a subject (e.g. `getName`), called **methods** or **class methods**.

Defining the *Developer* class once is just one part; instantiating it is the other. The class definition is the blueprint of its capabilities, and usage occurs when an instance is created with the `new` statement. If a JavaScript class definition exists, one can create *multiple* instances of it:

Code Playground

```
// class definition
class Developer { ... }

// class instantiation
const robin = new Developer('Robin', 'Wieruch');

console.log(robin.getName());
// "Robin Wieruch"

// another class instantiation
const dennis = new Developer('Dennis', 'Wieruch');
```

```
console.log(dennis.getName());  
// "Dennis Wieruch"
```

The concept of a class with definition and instantiation is similar to a React component, which also has only *one* component definition, but can have *multiple* component instances:

src/App.js

```
// definition of App component  
function App() {  
  return (  
    <div>  
      <h1>My Hacker Stories</h1>  
  
      <Search />  
  
      <hr />  
  
      {  
        /* creating an instance of List component */  
        <List />  
        /* creating another instance of List component */  
        <List />  
      }  
    </div>  
  );  
}  
  
// definition of List component  
function List() { ... }
```

Once we've defined a **component**, we can use it as an HTML **element** anywhere in our JSX. The element produces a **component instance** of your component, or in other words, the component gets instantiated. You can create as many component instances as you want. It's not much different from a JavaScript class definition and usage. However, technically a JavaScript class and React component are not the same, just their usage makes it convenient to demonstrate their similarities.

Exercises:

- Familiarize yourself with the terms *component definition*, *component instance*, and *element*.
- Experiment by creating multiple component instances of a List component.
- From the previous section: Ask yourself again what problems could arise if we keep treating the `list` variable as a global variable.

- Think about how it could be possible to give each List component its own `list` variable.
- Optional: [Rate this section](#)⁵⁷.

⁵⁷<https://forms.gle/sf1UMNR58v3NsRUSA>

React DOM

Now that we've learned about component definitions and their instantiation, we can move to the App component's instantiation. It has been in our application from the start, in the `src/index.js` file:

`src/index.js`

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

import App from './App';

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

Next to React which is imported from `react`, there is another imported library called `react-dom`, in which a `ReactDOM.render()` function uses an HTML node to replace it with JSX. Essentially that's everything needed to integrate React into any application which uses HTML. In more detail, `ReactDOM.render()` expects two arguments; the first is to render the JSX. It creates an instance of your App component, though it can also pass simple JSX without any component instantiation:

Code Playground

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

The second argument specifies where the React application enters your HTML. It expects an element with an `id='root'`, found in the `public/index.html` file. This is a basic HTML file.

Exercises:

- Open the `public/index.html` to see where the React application enters your HTML.
- Read more about [rendering elements in React](#)⁵⁸.
- Optional: [Rate this section](#)⁵⁹.

⁵⁸<https://bit.ly/3aUySgP>

⁵⁹<https://forms.gle/zSqHUhmsuQ35vqoj9>

React Component Definition (Advanced)

The following refactoring recommendations are optional to explain the different JavaScript/React patterns. You can build complete React applications without these advanced patterns, so don't be discouraged if they seem too complicated.

JavaScript has multiple ways to declare functions. So far, we have used the function statement, though arrow functions can be used more concisely:

Code Playground

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

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

You can remove the parentheses in an arrow function expression if it has only one argument, but multiple arguments require parentheses:

Code Playground

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

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

// not allowed
const item, index => { ... }

// allowed (recommended)
const (item, index) => { ... }
```

Defining React function components with arrow functions makes them more concise:

src/App.js

```
const App = () => {
  return (
    <div>
      ...
    </div>
  );
};

const Search = () => {
  return (
    <div>
      <label htmlFor="search">Search: </label>
      <input id="search" type="text" />
    </div>
  );
};

const List = () => {
  return (
    <ul>
      ...
    </ul>
  );
};
```

This also holds true for other functions, like the one we used in our JavaScript array's map method:

src/App.js

```
const List = () => {
  return (
    <ul>
      {list.map((item) => {
        return (
          <li 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>
          </li>
        );
      })}
    </ul>
  );
};
```

```
    );  
  }  
</ul>  
);  
};
```

If an arrow function's only purpose is to return a value and it doesn't have any business logic in between, you can remove the **block body** (curly braces) of the function. In a **concise body**, an **implicit return statement** is attached, so you can remove the return statement:

Code Playground

```
// with block body  
const countPlusOne = (count) => {  
  // perform any task in between  
  
  return count + 1;  
};  
  
// with concise body  
const countPlusOne = (count) =>  
  count + 1;  
  
// with concise body as one line  
const countPlusOne = (count) => count + 1;
```

This can be done for the App, List, and Search components as well, because they only return JSX and don't perform any task in between. Again it also applies to the arrow function that's used in the map function:

src/App.js

```
const App = () => (  
  <div>  
    ...  
  </div>  
)  
  
const Search = () => (  
  <div>  
    <label htmlFor="search">Search: </label>  
    <input id="search" type="text" />  
  </div>  
)
```

```

const List = () => (
  <ul>
    {list.map((item) => (
      <li 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>
      </li>
    ))}
  </ul>
);

```

Our JSX is more concise now, as it omits the function statement, the curly braces, and the return statement. However, remember this is an optional step, and that it's acceptable to use normal functions instead of arrow functions and block bodies with curly braces for arrow functions over implicit returns. Sometimes block bodies will be necessary to introduce more business logic between function signature and return statement:

Code Playground

```

const App = () => {
  // perform any task in between

  return (
    <div>
      ...
    </div>
  );
};

```

Be sure to understand this refactoring concept, because we'll move quickly from arrow function components with and without block bodies as we go. Which one we use will depend on the requirements of the component.

Exercises:

- Confirm your [source code](#)⁶⁰.
 - Confirm the [changes](#)⁶¹.
- Read more about [JavaScript arrow functions](#)⁶².

⁶⁰<https://bit.ly/2ZbLXQz>

⁶¹<https://bit.ly/3pkH2aS>

⁶²<https://mzl.la/3BYCOcp>

- Familiarize yourself with arrow functions with block body and return, and concise body without return.
- Optional: [Rate this section](#)⁶³.

⁶³<https://forms.gle/iWSchmqasbZUWUpT8>

Handler Function in JSX

The Search component still has the input field and label, which we haven't used. In HTML outside of JSX, input fields have an [onChange handler](#)⁶⁴. We're going to discover how to use onChange handlers with a React component's JSX. First, refactor the Search component from a concise body to a block body so we can add implementation details:

src/App.js

```
const Search = () => {
  // do something in between

  return (
    <div>
      <label htmlFor="search">Search: </label>
      <input id="search" type="text" />
    </div>
  );
};
```

Next, define a function – which can be a normal or arrow function – for the change event of the input field. In React, this function is called an **(event) handler**. Then the function can be passed to the onChange attribute (JSX named attribute) of the HTML input field:

src/App.js

```
const Search = () => {
  const handleChange = (event) => {
    console.log(event);
  };

  return (
    <div>
      <label htmlFor="search">Search: </label>
      <input id="search" type="text" onChange={handleChange} />
    </div>
  );
};
```

After opening your application in a web browser, open the browser's developer tools to see logging occur after you type into the input field. What you see is called a **synthetic event** defined by a JavaScript object. Through this object, we can access the emitted value of the input field:

⁶⁴<https://mzl.la/3n9wit4>

src/App.js

```
const Search = () => {
  const handleChange = (event) => {
    console.log(event.target.value);
  };

  return ( ... );
};
```

React's synthetic event is essentially a wrapper around the [browser's native event](#)⁶⁵, with more functions that are useful to prevent native browser behavior (e.g. refreshing a page after the user clicks a form's submit button). Sometimes you will use the event, sometimes you won't need it.

After all, this is how we give HTML elements in JSX handler functions to respond to user interaction. Always pass functions to these handlers, not the return value of the function, except when the return value is a function. Knowing this is crucial because it's a well-known source for bugs in a React beginners application:

Code Playground

```
// don't do this
<input
  id="search"
  type="text"
  onChange={handleChange()}
/>

// do this instead
<input
  id="search"
  type="text"
  onChange={handleChange}
/>
```

As you can see, HTML and JavaScript work well together in JSX. JavaScript in HTML can display JavaScript variables (e.g. `title` JavaScript string in `{title}`), can pass JavaScript primitives to HTML attributes (e.g. `url` JavaScript string to `` HTML element), and can pass functions to an HTML element's attributes for handling user interactions.

Exercises:

- Confirm your [source code](#)⁶⁶.
 - Confirm the [changes](#)⁶⁷.

⁶⁵<https://mzl.la/30Dk8kt>

⁶⁶<https://bit.ly/3lY8usB>

⁶⁷<https://bit.ly/3BYqQzp>

- Read more about [React's event handler](#)⁶⁸ and [React's events](#)⁶⁹.
- Optional: [Rate this section](#)⁷⁰.

⁶⁸<https://www.robinwieruch.de/react-event-handler>

⁶⁹<https://bit.ly/3jiFdaz>

⁷⁰<https://forms.gle/oSKyMudmb8X1iSsv8>

React Props

We are currently using the `list` variable as a global variable in the current application. We used it directly from the global scope in the `App` component, and later in the `List` component. This could work if you only had one global variable, but it isn't maintainable with multiple variables across multiple components. By using so-called props in React, we can pass variables as information from one component to another component.

Before using props for the first time, we'll move the `list` from the global scope into the `App` component and rename it to its actual domain. Don't forget to refactor the `App` component's function from concise to block body in order to define the `list` in between:

`src/App.js`

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

  return ( ... );
};
```

Next, we'll use **React props** to pass the list of items to the `List` component. The variable is called `stories` in the `App` component, and we pass it under this name to the `List` component. In the `List` component's instantiation, however, it is assigned to the `list` HTML attribute:

src/App.js

```
const App = () => {
  const stories = [ ... ];

  return (
    <div>
      <h1>My Hacker Stories</h1>

      <Search />

      <hr />

      <List list={stories} />
    </div>
  );
};
```

After passing it to the List component, we can access it as `list` property from the `props` object in the List component's function signature:

src/App.js

```
const List = (props) => (
  <ul>
    {props.list.map((item) => (
      <li 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>
      </li>
    ))}
  </ul>
);
```

Everything that we pass from a parent component to a child component via a component element's HTML attribute can be accessed in the child component. The child component receives an object parameter (`props`) which includes all the passed attributes as properties (`props`).

Using this mechanism of passing information from one component down to another with React `props`, we've prevented the `list` variable from polluting the global scope in our `src/App.js` file when

we defined it outside of the component. Now, the list is defined as `stories` in our `App` component. However, since `stories` is not used in the `App` component directly, but in one of its child components, we passed it as props to the `List` component. We could also define `stories` directly in the `List` component and would not need to use props in the first place, however, in the future we will make use of the `stories` in the `App` component and thus will keep it there.

Another use case for React props is the `List` component and its potential child component. Previously, we couldn't extract an `Item` component from the `List` component, because we didn't know how to pass each item to the extracted `Item` component. With the new knowledge about React props, we can perform the component extraction and pass each item along to the `List` component's new child component:

`src/App.js`

```
const List = (props) => (  
  <ul>  
    {props.list.map((item) => (  
      <Item key={item.objectID} item={item} />  
    ))}  
  </ul>  
);  
  
const Item = (props) => (  
  <li>  
    <span>  
      <a href={props.item.url}>{props.item.title}</a>  
    </span>  
    <span>{props.item.author}</span>  
    <span>{props.item.num_comments}</span>  
    <span>{props.item.points}</span>  
  </li>  
);
```

In conclusion, one can see how props are used to pass information down the component tree. Following this explanation, information (props) can only be passed from a parent to a child component and not vice versa. We will learn how to overcome this limitation later.

Exercises:

- Confirm your [source code](#)⁷¹.
 - Confirm the [changes](#)⁷².
- Read more about [how to pass props to React components](#)⁷³.
- Optional: [Rate this section](#)⁷⁴.

⁷¹<https://bit.ly/3jlmUBz>

⁷²<https://bit.ly/3lW2sIX>

⁷³<https://www.robinwieruch.de/react-pass-props-to-component>

⁷⁴<https://forms.gle/APwaUSAuVAAA56sY6>

React State

In contrast to React props, **React state** is used to make applications interactive. Both concepts, props and state, have clear defined purposes: Props are used to pass information down the component tree, state is used to alter information over time. Both can work hand in hand as well. We will see what this means in the following sections.

Let's start with state in React with the following use case: Whenever a user types something into an HTML input field, the user may want to see this typed information (state) displayed somewhere else in the application. Therefore we need some way to change information over time and, what's more important, to notify React to re-render its component(s) again. A naive (but wrong) approach would be the following:

src/App.js

```
const Search = () => {
  let searchTerm = '';

  const handleChange = (event) => {
    searchTerm = event.target.value;
  };

  return (
    <div>
      <label htmlFor="search">Search: </label>
      <input id="search" type="text" onChange={handleChange} />

      <p>
        Searching for <strong>{searchTerm}</strong>.
      </p>
    </div>
  );
};
```

When you try this in the browser, you will see that the output does not appear below the HTML input field after typing into it. However, this approach is not too far away from the actual solution. What's missing after all is the mechanisms to notify React to re-render the component with the new `searchTerm` state after the event handler updated it. In order to do so, we need to tell React that `searchTerm` is a state that changes over time and that whenever it changes React has to re-render its affected component(s). Fortunately, React offers us a utility function called `useState` for it:

src/App.js

```
const Search = () => {  
  const [searchTerm, setSearchTerm] = React.useState('');  
  
  ...  
};
```

React's `useState` function takes an *initial state* as an argument – where we will use an empty string. By providing this initial state to `useState`, we are telling React that this state will change over time. Furthermore, calling this function will return an array with two entries: The first entry (`searchTerm`) represents the *current state*; the second entry is a *function to update this state* (`setSearchTerm`). I will sometimes refer to this function as *state updater function*. Both entries are everything we need to display the current state and to alter the current state:

src/App.js

```
const Search = () => {  
  const [searchTerm, setSearchTerm] = React.useState('');  
  
  const handleChange = (event) => {  
    setSearchTerm(event.target.value);  
  };  
  
  return (  
    <div>  
      <label htmlFor="search">Search: </label>  
      <input id="search" type="text" onChange={handleChange} />  
  
      <p>  
        Searching for <strong>{searchTerm}</strong>.  
      </p>  
    </div>  
  );  
};
```

When the user types into the input field, the input field's change event is captured by the event handler. The handler's logic uses the event's value and the state updater function to set the updated state. After the updated state is set in a component, the component renders again, meaning the component function runs again. The updated state becomes the current state and is displayed in the component's JSX.

As an exercise, put a `console.log()` into each of your components. For example, the App component gets a `console.log('App renders')`, the List component gets a `console.log('List renders')` and

so on. Now check your browser: For the first rendering, all loggings should appear, however, once you type into the HTML input field, only the Search component's logging should appear. That's because React only re-renders this component (and all of its child components), because its state changed.

It's important to note that the `useState` function is called a **React hook**. It's only one of many hooks provided by React and this section only scratched the surface. You will learn more about them throughout the next sections.

Exercises:

- Confirm your [source code](#)⁷⁵.
 - Confirm the [changes](#)⁷⁶.
- Read more about [JavaScript array destructuring](#)⁷⁷.
- Read more about [React's useState Hook](#)⁷⁸.
- Optional: [Rate this section](#)⁷⁹.

⁷⁵<https://bit.ly/3prVjSO>

⁷⁶<https://bit.ly/30ISOBv>

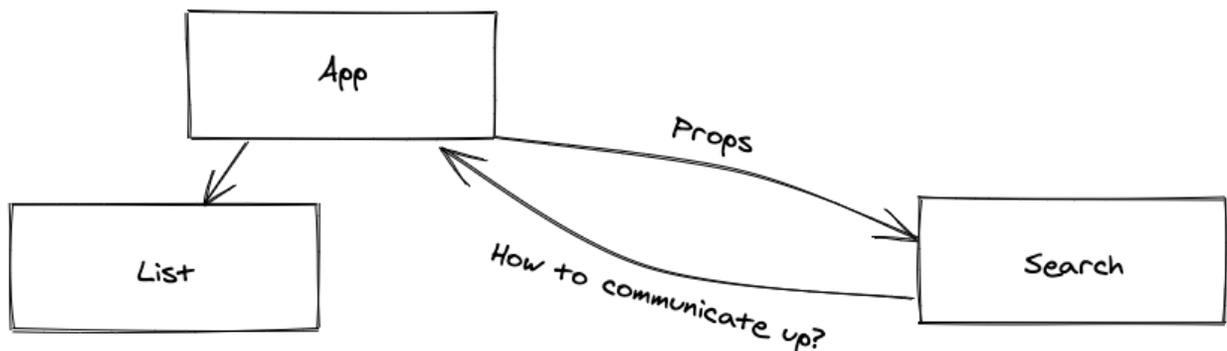
⁷⁷<https://mzl.la/3ncC7WI>

⁷⁸<https://www.robinwieruch.de/react-usestate-hook>

⁷⁹<https://forms.gle/ZJNbQqq3Lw3RiD4H9>

Callback Handlers in JSX

The last sections taught us important lessons about props and state in React. While props are passed down as information from parent to child components, state can be used to change information over time and to make React show this changed information. However, we don't have all the pieces yet to make our components talk to each other. At the moment, the Search component does not share its state with other components, so it's only used and updated by the Search component and thus becomes useless for the other components.



There is no way to pass information up the component tree, since props are naturally only passed downwards. However, we can introduce a **callback handler**: A callback function gets introduced (A), is used elsewhere (B), but “calls back” to the place it was introduced (C):

src/App.js

```
const App = () => {
  const stories = [ ... ];

  // A
  const handleSearch = (event) => {
    // C
    console.log(event.target.value);
  };

  return (
    <div>
      <h1>My Hacker Stories</h1>

      {/* // B */}
      <Search onSearch={handleSearch} />

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

```
        <List list={stories} />
      </div>
    );
  };
```

Now our Search component can use this callback handler from its incoming props to call it whenever a user types into the HTML input field:

src/App.js

```
const Search = (props) => {
  const [searchTerm, setSearchTerm] = React.useState('');

  const handleChange = (event) => {
    setSearchTerm(event.target.value);

    // B
    props.onSearch(event);
  };

  return ( ... );
};
```

The concept of the callback handler in a nutshell: We pass a function from a parent component (App) to a child component (Search) via props; we call this function in the child component (Search), but have the actual implementation of the called function in the parent component (App). Essentially when an (event) handler is passed as props from a parent component to its child component, it becomes a callback handler. React props are always passed down the component tree, and callback handlers passed as functions in props can be used to communicate up the component hierarchy.

Exercises:

- Confirm your [source code](#)⁸⁰.
 - Confirm the [changes](#)⁸¹.
- Revisit the concepts of handler and callback handler as many times as you need.
- Optional: [Rate this section](#)⁸².

⁸⁰<https://bit.ly/3DUWm1O>

⁸¹<https://bit.ly/3jizj9s>

⁸²<https://forms.gle/3LoBoWKCmnt2YpnA7>

Lifting State in React

In this section, we are confronted with the following task: Use the `searchTerm` from the `Search` component to filter the `stories` by title in the `App` component before they reach the `List` component. In the last section, we established a callback handler to pass information from the `Search` component up to the `App` component. However, if you check the code, it seems somehow difficult to access the value coming from the callback handler's event in the `App` component in order to filter the list. The incoming value is only accessible in the callback handler's function. We need to figure out how to share the `Search` component's state across multiple components (`App`, `List`) who are interested in it. Therefore, we'll need to **lift state up** from `Search` to `App` component to share the state with more components:

`src/App.js`

```
const App = () => {
  const stories = [ ... ];

  const [searchTerm, setSearchTerm] = React.useState('');

  const handleSearch = (event) => {
    setSearchTerm(event.target.value);
  };

  return (
    <div>
      <h1>My Hacker Stories</h1>

      <Search onSearch={handleSearch} />

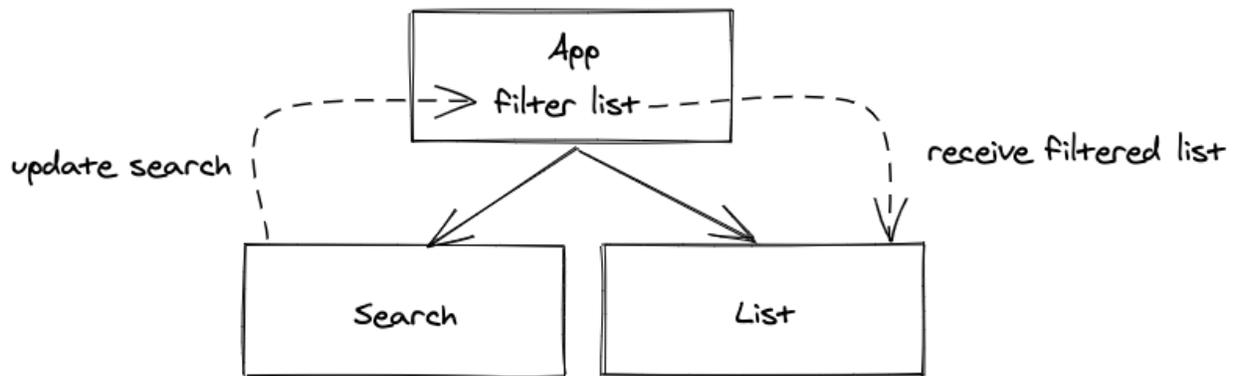
      <hr />

      <List list={stories} />
    </div>
  );
};

const Search = (props) => (
  <div>
    <label htmlFor="search">Search: </label>
    <input id="search" type="text" onChange={props.onSearch} />
  </div>
);
```

We learned about the callback handler previously, because it helps us to keep an open communication channel from Search to App component. Now the Search component doesn't manage the state anymore, but only passes up the event to the App component via a callback handler after the text is entered into the HTML input field. You could also display the `searchTerm` again in the App component (from state, when using `searchTerm` directly) or Search component (from props, when passing the `searchTerm` state down as props).

Rule of thumb: Always manage state at a component level where every component that's interested in it is one that either manages the state (using information directly from state, e.g. App component) or a component below the managing component (using information from props, e.g. List or Search). If a component below needs to update the state (e.g. Search), pass a callback handler down to it which allows it to update it. If a component needs to use the state (e.g. displaying it), pass it down as props.



Finally, by managing the search feature's state in the App component, we can filter the stories with the stateful `searchTerm` before passing them as `list` to the List component:

src/App.js

```

const App = () => {
  const stories = [ ... ];

  const [searchTerm, setSearchTerm] = React.useState('');

  const handleSearch = (event) => {
    setSearchTerm(event.target.value);
  };

  const searchedStories = stories.filter(function (story) {
    return story.title.includes(searchTerm);
  });

  return (
    <div>

```

```
    <h1>My Hacker Stories</h1>

    <Search onSearch={handleSearch} />

    <hr />

    <List list={searchedStories} />
  </div>
);
};
```

Here, the [JavaScript array's built-in filter function](#)⁸³ is used to create a new filtered array. The filter function takes a function as an argument, which accesses each item in the array and returns true or false. If the function returns true, meaning the condition is met, the item stays in the newly created array; if the function returns false, it's removed:

Code Playground

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

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

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

The filter function can be made more concise by using an arrow function with an immediate return:

⁸³<https://mzl.la/3BYFAOR>

src/App.js

```
const App = () => {  
  ...  
  
  const searchedStories = stories.filter((story) =>  
    story.title.includes(searchTerm)  
  );  
  
  ...  
};
```

That’s all to the refactoring steps of the inlined function for the filter function. There are many variations to it – and it’s not always simple to keep a good balance between readable and conciseness – however, I feel like keeping it concise whenever possible keeps it most of the time readable as well.

What’s not working very well yet: The filter function checks whether the `searchTerm` is present in our story item’s title, but it’s still too opinionated about the letter case. If we search for “react”, there is no filtered “React” story in your rendered list. To fix this problem, we have to lower case the story’s title and the `searchTerm` to make them equal.

src/App.js

```
const App = () => {  
  ...  
  
  const searchedStories = stories.filter((story) =>  
    story.title.toLowerCase().includes(searchTerm.toLowerCase())  
  );  
  
  ...  
};
```

Now you should be able to search for “eact”, “React”, or “react” and see one of two displayed stories. Congratulations, you have just added your first real interactive feature to your application by leveraging state – to derive a filtered list of stories – and props – by passing a callback handler to the Search component.

Exercises:

- Confirm your [source code](#)⁸⁴.
 - Confirm the [changes](#)⁸⁵.

⁸⁴<https://bit.ly/3vtfBwo>

⁸⁵<https://bit.ly/3DSiuK6>

- Read more about [lifting state in React](#)⁸⁶.
- Optional: [Rate this section](#)⁸⁷.

⁸⁶<https://www.robinwieruch.de/react-lift-state>

⁸⁷<https://forms.gle/EqJGjxCM1Xzw9S6g7>

React Controlled Components

When you type into your HTML input field and see the characters showing up, you may have noticed that the element itself holds an internal state, because we are not providing any external value to it. Let me show you where this behavior leads to unexpected results: After applying the following change – giving the `searchTerm` an initial state of ‘React’ – can you spot the mistake in your browser?

src/App.js

```
const App = () => {  
  const stories = [ ... ];  
  
  const [searchTerm, setSearchTerm] = React.useState('React');  
  
  ...  
};
```

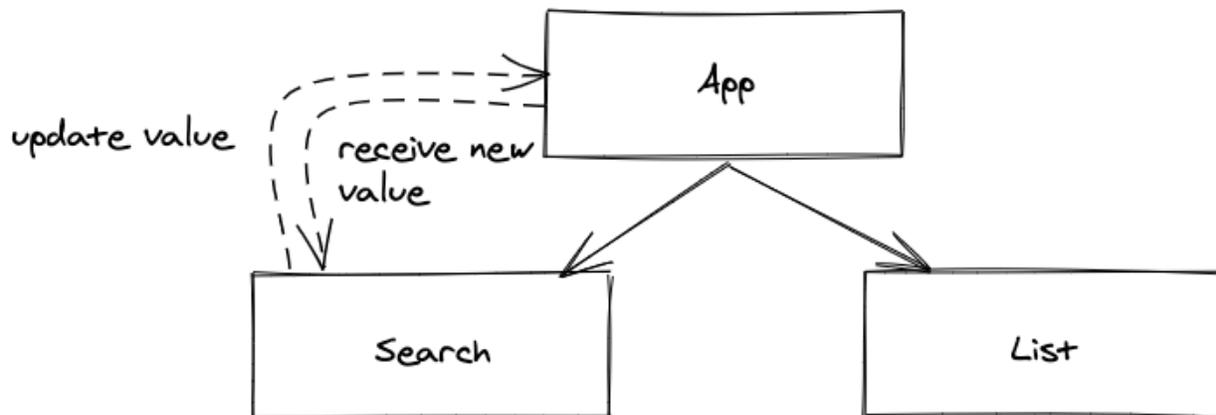
While the list has been filtered respectively to this search term, the HTML input field doesn't show the value. Only when typing into the input field we see the change reflected in it. However, if we want to start properly with the initial state in the input field, we need to convert the Search component with its input field into a so-called **controlled component**. So far, the input field doesn't know anything about the `searchTerm`. It only uses the `onChange` handler to inform us of a change. Good for us that the input field has a `value` attribute which we can use as well:

src/App.js

```
const App = () => {  
  const stories = [ ... ];  
  
  const [searchTerm, setSearchTerm] = React.useState('React');  
  
  ...  
  
  return (  
    <div>  
      <h1>My Hacker Stories</h1>  
  
      <Search search={searchTerm} onSearch={handleSearch} />  
  
      ...  
    </div>  
  );  
};
```

```
const Search = (props) => (
  <div>
    <label htmlFor="search">Search: </label>
    <input
      id="search"
      type="text"
      value={props.search}
      onChange={props.onSearch}
    />
  </div>
);
```

Now the input field uses the correct initial value when displaying it in the browser. When we use the `searchTerm` state from the `App` component via props, we force the input field to use this value over its internally managed element's state.



We learned about controlled components in this section. Taking all the previous sections as learning steps into consideration, we discovered another concept called **unidirectional data flow**:

Visualization

UI -> Side-Effect -> State -> UI -> ...

A React application and its components start with an initial state, which may or may not be passed down as props to interested child components. It's rendered for the first time as a UI. Once a side-effect occurs, like user interaction (e.g. typing into an input field) or data loading from a remote API, the change is captured in React's state either in the component itself or by notifying parent components via a callback handler. Once state has been changed, all components below the component with the modified state are re-rendered (meaning: the component functions run again).

In the previous sections, we also learned about React's **component lifecycle**. At first, all components are instantiated from the top to the bottom of the component hierarchy. This includes all hooks (e.g. `useState`) that are instantiated with their initial values (e.g. initial state). From there, the UI awaits side-effects like user interactions. Once the state is changed (e.g. current state changed via state updater function from `useState`), all components below render again.

Every run through a component's function takes the *recent value* (e.g. current state) from React's `useState` Hook and *doesn't* reinitialize them again (e.g. initial state). This might seem odd, as one could assume the `useState` hooks function re-initializes again with its initial value, but it doesn't. Hooks initialize only once when the component renders for the first time, after which React tracks them internally with their most recent values.

Exercises:

- Confirm your [source code](#)⁸⁸.
 - Confirm the [changes](#)⁸⁹.
- Read more about [controlled components in React](#)⁹⁰.
- Experiment with `console.log()` in your React components and observe how your changes render, both initially and after the input field changes.
- Optional: [Rate this section](#)⁹¹.

⁸⁸<https://bit.ly/3aXr7GZ>

⁸⁹<https://bit.ly/3aV4XVO>

⁹⁰<https://www.robinwieruch.de/react-controlled-components/>

⁹¹<https://forms.gle/7VYTww2EQiPkFnaR8>

Props Handling (Advanced)

Props are passed from parent to child down the component tree. Since we use props to transport information from component to component frequently, and sometimes via other components which are in between, it is useful to know a few tricks to make passing props more convenient. Note that the following refactorings are recommended for you to learn different JavaScript/React patterns, though you can still build complete React applications without them. Consider these advanced React techniques that will make your source code more concise.

Object Destructuring

After all, React props are a JavaScript object, else we couldn't access `props.list` or `props.onSearch` in our React components. Since `props` is an object which just passes information from one component to another component, we can apply a couple JavaScript tricks to it. For example, accessing an object's properties with modern [JavaScript object destructuring](#)⁹²:

Code Playground

```
const user = {
  firstName: 'Robin',
  lastName: 'Wieruch',
};

// without object destructuring
const firstName = user.firstName;
const lastName = user.lastName;

console.log(firstName + ' ' + lastName);
// "Robin Wieruch"

// with object destructuring
const { firstName, lastName } = user;

console.log(firstName + ' ' + lastName);
// "Robin Wieruch"
```

If we need to access multiple properties of an object, using one line of code instead of multiple lines is often simpler and more elegant. That's why object destructuring is already widely used in JavaScript. Let's transfer this knowledge to the React props in our Search component. First, we have to refactor the Search component's arrow function from the concise body into block body:

⁹²<https://mzl.la/30KbXTC>

src/App.js

```
const Search = (props) => {
  return (
    <div>
      <label htmlFor="search">Search: </label>
      <input
        id="search"
        type="text"
        value={props.search}
        onChange={props.onSearch}
      />
    </div>
  );
};
```

And second, we can apply the destructuring of the props object in the component's function body:

src/App.js

```
const Search = (props) => {
  const { search, onSearch } = props;

  return (
    <div>
      <label htmlFor="search">Search: </label>
      <input
        id="search"
        type="text"
        value={search}
        onChange={onSearch}
      />
    </div>
  );
};
```

That's a basic destructuring of the props object in a React component, so that the object's properties can be used conveniently in the component. However, we also had to refactor the Search component's arrow function from concise body into block body to access the properties of props with the object destructuring in the function's body. This would happen quite often if we followed this pattern, and it wouldn't make things easier for us, because we would constantly have to refactor our components. We can take all this one step further by destructuring the props object right away in the function signature of our component, omitting the function's block body of the component again:

src/App.js

```
const Search = ({ search, onSearch }) => (  
  <div>  
    <label htmlFor="search">Search: </label>  
    <input  
      id="search"  
      type="text"  
      value={search}  
      onChange={onSearch}  
    />  
  </div>  
);
```

React's props are rarely used in components by themselves; rather, all the information that is contained in the props object is used. By destructuring the props object right away in the component's function signature, we can conveniently access all information without dealing with its props container. The List and Item components can perform the same props destructuring:

src/App.js

```
const List = ({ list }) => (  
  <ul>  
    {list.map((item) => (  
      <Item key={item.objectID} item={item} />  
    ))}  
  </ul>  
);  
  
const Item = ({ item }) => (  
  <li>  
    <span>  
      <a href={item.url}>{item.title}</a>  
    </span>  
    <span>{item.author}</span>  
    <span>{item.num_comments}</span>  
    <span>{item.points}</span>  
  </li>  
);
```

This should be the basic lesson learned from this section, however, we can take this one step further with the following advanced lessons. The incoming `item` parameter in the Item component has something in common with the previously discussed `props` parameter: they are both JavaScript

objects. Also, even though the `item` object has already been destructured from the `props` in the `Item` component's function signature, it isn't directly used in the `Item` component. The `item` object only passes its information (object properties) to the elements.

Nested Destructuring

The current solution is fine as you will see in the ongoing discussion. However, I want to show you two more variations of it, because there are many things to learn about JavaScript objects in React here. Let's get started with *nested destructuring* and how it works:

Code Playground

```
const user = {
  firstName: 'Robin',
  pet: {
    name: 'Trixi',
  },
};

// without object destructuring
const firstName = user.firstName;
const name = user.pet.name;

console.log(firstName + ' has a pet called ' + name);
// "Robin has a pet called Trixi"

// with nested object destructuring
const {
  firstName,
  pet: {
    name,
  },
} = user;

console.log(firstName + ' has a pet called ' + name);
// "Robin has a pet called Trixi"
```

The nested destructuring helps us to gather all the needed information of the `item` object in the function signature for its immediate usage in the component's elements. However, nested destructuring introduces lots of clutter through indentations in the function signature. Even though it's not the most readable option here, note that it can still be useful in other scenarios.

src/App.js

```
// Variation 1: Nested Destructuring
```

```
const Item = ({
  item: {
    title,
    url,
    author,
    num_comments,
    points,
  },
}) => (
  <li>
    <span>
      <a href={url}>{title}</a>
    </span>
    <span>{author}</span>
    <span>{num_comments}</span>
    <span>{points}</span>
  </li>
);
```

The nested destructuring helps us to gather all the needed information of the `item` object in the function signature for its immediate usage in the component's elements. However, nested destructuring introduces lots of clutter through indentations in the function signature. While it's here not the most readable option, it can be useful in other scenarios though.

Spread and Rest Operators

Let's take another approach with JavaScript's spread and rest operators. In order to prepare for it, we will refactor our `List` and `Item` components to the following implementation. Rather than passing the `item` as an object from `List` to `Item` component, we are passing every property of the `item` object:

src/App.js

```
// Variation 2: Spread and Rest Operators
// 1. Step

const List = ({ list }) => (
  <ul>
    {list.map((item) => (
      <Item
        key={item.objectID}
        title={item.title}
        url={item.url}
        author={item.author}
        num_comments={item.num_comments}
        points={item.points}
      />
    ))}
  </ul>
);

const Item = ({ title, url, author, num_comments, points }) => (
  <li>
    <span>
      <a href={url}>{title}</a>
    </span>
    <span>{author}</span>
    <span>{num_comments}</span>
    <span>{points}</span>
  </li>
);
```

Now, even though the Item component's function signature is more concise, the clutter ended up in the List component instead, because every property is passed to the Item component individually. We can improve this approach using [JavaScript's spread operator](https://mzl.la/3jetlkn)⁹³:

⁹³<https://mzl.la/3jetlkn>

Code Playground

```
const profile = {
  firstName: 'Robin',
  lastName: 'Wieruch',
};

const address = {
  country: 'Germany',
  city: 'Berlin',
};

const user = {
  ...profile,
  gender: 'male',
  ...address,
};

console.log(user);
// {
//   firstName: "Robin",
//   lastName: "Wieruch",
//   gender: "male"
//   country: "Germany",
//   city: "Berlin",
// }
```

JavaScript's spread operator allows us to literally spread all key/value pairs of an object to another object. This can also be done in React's JSX. Instead of passing each property one at a time via props from List to Item component as before, we can use JavaScript's spread operator to pass all the object's key/value pairs as attribute/value pairs to a JSX element:

src/App.js

```
// Variation 2: Spread and Rest Operators
// 2. Step

const List = ({ list }) => (
  <ul>
    {list.map((item) => (
      <Item key={item.objectID} {...item} />
    ))}
  </ul>
);
```

```
const Item = ({ title, url, author, num_comments, points }) => (  
  <li>  
    <span>  
      <a href={url}>{title}</a>  
    </span>  
    <span>{author}</span>  
    <span>{num_comments}</span>  
    <span>{points}</span>  
  </li>  
)
```

This refactoring made the process of passing the information from List to Item component more concise. Finally, we'll use [JavaScript's rest parameters](#)⁹⁴ as the icing on the cake. The JavaScript rest operator happens always as the last part of an object destructuring:

Code Playground

```
const user = {  
  id: '1',  
  firstName: 'Robin',  
  lastName: 'Wieruch',  
  country: 'Germany',  
  city: 'Berlin',  
};  
  
const { id, country, city, ...userWithoutAddress } = user;  
  
console.log(userWithoutAddress);  
// {  
//   firstName: "Robin",  
//   lastName: "Wieruch"  
// }  
  
console.log(id);  
// "1"  
  
console.log(city);  
// "Berlin"
```

Even though both have the same syntax (three dots), the rest operator shouldn't be mistaken with the spread operator. Whereas the rest operator happens on the left side of an assignment, the spread

⁹⁴<https://mzl.la/3GeJbef>

operator happens on the right side. The rest operator is always used to separate an object from some of its properties.

Now it can be used in our List component to separate the `objectID` from the `item`, because the `objectID` is only used as a key and isn't used in the Item component. Only the remaining (rest) item gets spread as attribute/value pairs into the Item component (as before):

src/App.js

```
// Variation 2: Spread and Rest Operators
// Final Step

const List = ({ list }) => (
  <ul>
    {list.map(({ objectID, ...item }) => (
      <Item key={objectID} {...item} />
    ))}
  </ul>
);

const Item = ({ title, url, author, num_comments, points }) => (
  <li>
    <span>
      <a href={url}>{title}</a>
    </span>
    <span>{author}</span>
    <span>{num_comments}</span>
    <span>{points}</span>
  </li>
);
```

In this final variation, the rest operator is used to destructure the `objectID` from the rest of the `item` object. Afterward, the `item` is spread with its key/values pairs into the Item component. While this final variation is very concise, it comes with advanced JavaScript features that may be unknown to some.

In this section, we have learned about JavaScript object destructuring which can be commonly used not only for the `props` object, but also for other objects like the `item` object. We have also seen how nested destructuring can be used (Variation 1), but also how it didn't add any benefits in our case because it just made the component bigger. In the future, you are more likely to find use cases for nested destructuring which are beneficial. Last but not least, you have learned about JavaScript's spread and rest operators, which shouldn't be confused with each other, to perform operations on JavaScript objects and to pass the `props` object from one component to another component in the most concise way. In the end, I want to point out the initial version again which we will keep over for the next sections:

src/App.js

```
const List = ({ list }) => (  
  <ul>  
    {list.map((item) => (  
      <Item key={item.objectID} item={item} />  
    ))}  
  </ul>  
);  
  
const Item = ({ item }) => (  
  <li>  
    <span>  
      <a href={item.url}>{item.title}</a>  
    </span>  
    <span>{item.author}</span>  
    <span>{item.num_comments}</span>  
    <span>{item.points}</span>  
  </li>  
);
```

It may not be the most concise, but it is the easiest to reason about. Variation 1 with its nested destructuring didn't add much benefit and variation 2 may add too many advanced JavaScript features (spread operator, rest operator) which are not familiar to everyone. After all, all these variations have their pros and cons. When refactoring a component, always aim for readability, especially when working in a team of people, and make sure they're using a common React code style.

Exercises:

- Confirm your [source code](#)⁹⁵.
 - Confirm the [changes](#)⁹⁶.
- Read more about [JavaScript's destructuring assignment](#)⁹⁷.
- Think about the difference between JavaScript array destructuring – which we used for React's `useState` hook – and object destructuring.
- Read more about [JavaScript's spread operator](#)⁹⁸.
- Read more about [JavaScript's rest parameters](#)⁹⁹.
- Get a good sense about JavaScript (e.g. destructuring, spread operator, rest parameters) and what's related to React (e.g. props) from the last lessons.

⁹⁵<https://bit.ly/30IuiAu>

⁹⁶<https://bit.ly/3G45eUI>

⁹⁷<https://mzl.la/30KbXTC>

⁹⁸<https://mzl.la/3jetlkn>

⁹⁹<https://mzl.la/3GeJbef>

- Continue to use your favorite way to handle React's props. If you're still undecided, consider the variation used in the previous section.
- Optional: [Rate this section](#)¹⁰⁰.

¹⁰⁰<https://forms.gle/WNB4R6yEP1hot3tK8>

React Side-Effects

A React component's returned output is defined by its props and state. In contrast, side-effects don't change this output directly (but can change it indirectly). They are used to interact with APIs outside of the component (e.g. browser's localStorage API, remote APIs for data fetching), measuring HTML element's width and height, or setting timers in JavaScript. These are only a few examples of side-effects in React components and we will get to know one of them in this section.

Wouldn't it be great if our Search component could remember the most recent search, so that the application opens it in the browser whenever it restarts? Let's implement this feature by using a side-effect to store the recent search from the browser's local storage and load it upon component initialization. First, use the local storage to store the searchTerm accompanied by an identifier whenever a user types into the HTML input field:

src/App.js

```
const App = () => {  
  ...  
  
  const handleSearch = (event) => {  
    setSearchTerm(event.target.value);  
  
    localStorage.setItem('search', event.target.value);  
  };  
  
  ...  
};
```

Second, use the stored value, if a value exists, to set the initial state of the searchTerm in React's useState Hook. Otherwise, default to our initial state (here "React") as before:

src/App.js

```
const App = () => {  
  ...  
  
  const [searchTerm, setSearchTerm] = React.useState(  
    localStorage.getItem('search') || 'React'  
  );  
  
  ...  
};
```

JavaScript's logical OR operator `||`¹⁰¹ returns the truthy operand in this expression and is short-circuited if `localStorage.getItem('search')` returns a truthy value. It's used as a shorthand for the following implementation:

Code Playground

```
let hasStored;
if (localStorage.getItem('search')) {
  hasStored = true;
} else {
  hasStored = false;
}

const initialState = hasStored
  ? localStorage.getItem('search')
  : 'React';
```

When using the input field and refreshing the browser tab, the browser should remember the latest search term now. The feature is complete, but there is one flaw that may introduce bugs in the long run: The handler function should mostly be concerned with updating the state, but now it has this side-effect. If we use the `setSearchTerm` function elsewhere in our application, we may break the feature we implemented because we cannot enforce that the local storage will also get updated. Let's fix this by handling the side-effect at a centralized place. We'll use React's `useEffect` Hook to trigger the side-effect each time the `searchTerm` changes:

src/App.js

```
const App = () => {
  ...

  const [searchTerm, setSearchTerm] = React.useState(
    localStorage.getItem('search') || 'React'
  );

  React.useEffect(() => {
    localStorage.setItem('search', searchTerm);
  }, [searchTerm]);

  const handleSearch = (event) => {
    setSearchTerm(event.target.value);
  };

  ...
};
```

¹⁰¹<https://mzl.la/3aXxyrd>

React's `useEffect` Hook takes two arguments: The first argument is a function that runs our side-effect. In our case, the side-effect stores `searchTerm` into the browser's local storage. The second argument is a dependency array of variables. If one of these variables changes, the function for the side-effect is called. In our case, the function is called every time the `searchTerm` changes (e.g. when a user types into the HTML input field); and it's also called initially when the component renders for the first time.

Leaving out the second argument (the dependency array) would make the function for the side-effect run on every render (initial render and update renders) of the component. If the dependency array of React's `useEffect` is an empty array, the function for the side-effect is only called once, after the component renders for the first time. After all, the hook lets us opt into React's component lifecycle. It can be triggered when the component is first mounted, but also if one of its dependencies is updated.

In conclusion, using React `useEffect` Hook instead of managing the side-effect in the (event) handler has made the application more robust. *Whenever* and *wherever* the `searchTerm` state is updated via `setSearchTerm`, the browser's local storage will always be in sync with it.

Exercises:

- Confirm your [source code](#)¹⁰².
 - Confirm the [changes](#)¹⁰³.
- Read more about [React's `useEffect` Hook](#)¹⁰⁴.
- Read more about [using local storage with React](#)¹⁰⁵.
- Give the first argument's function a `console.log()` and experiment with React's `useEffect` Hook's dependency array. Check the logs for an empty dependency array too.
- Try the following scenario: In your browser, backspace the search term from the input field until nothing is left there. Internally, it should be set to an empty string now. Next, refresh the browser and check what it displays. You may be wondering why it does show "React" instead of "", because "" should be the recent search. That's because JavaScript's logical OR evaluates "" to false and thus takes "React" as the true value. If you want to prevent this and evaluate "" as true instead, you may want to exchange JavaScript's logical OR operator `||` with [JavaScript's nullish coalescing operator `??`](#)¹⁰⁶.
- Optional: [Rate this section](#)¹⁰⁷.

¹⁰²<https://bit.ly/3jj9TbC>

¹⁰³<https://bit.ly/3E12iGK>

¹⁰⁴<https://www.robinwieruch.de/react-useeffect-hook>

¹⁰⁵<https://www.robinwieruch.de/local-storage-react>

¹⁰⁶<https://mzl.la/2Z4bsU4>

¹⁰⁷<https://forms.gle/iCtVZHYt2XRnFAcBA>

React Custom Hooks (Advanced)

Thus far we've covered the two most popular hooks in React: `useState` and `useEffect`. `useState` is used for variables that change over time; `useEffect` is used to opt into the lifecycle of your components to introduce side-effects. We'll eventually cover more hooks that come with React, but next, we'll tackle **React custom Hooks**; that is, building a hook yourself.

We will use the two hooks we already possess to create a new custom hook called `useSemiPersistentState`, named as such because it manages state yet synchronizes with the local storage. It's not fully persistent because clearing the local storage of the browser deletes relevant data for this application. We will start with how we want to use the hook in our App component:

src/App.js

```
const App = () => {
  const stories = [ ... ];

  const [searchTerm, setSearchTerm] = useSemiPersistentState('React');

  const handleSearch = (event) => {
    setSearchTerm(event.target.value);
  };

  const searchedStories = stories.filter((story) =>
    story.title.toLowerCase().includes(searchTerm.toLowerCase())
  );

  return (
    ...
  );
};
```

Instead of using React's built-in `useState` Hook, we want to use this custom hook now. Under the hood, we want that this hook synchronizes the state with the browser's local storage. If you look closely at the App component now, you can see that none of the previously introduced local storage features are there anymore. That's because we will copy this functionality over to our new custom hook:

src/App.js

```
const useSemiPersistentState = () => {
  const [searchTerm, setSearchTerm] = React.useState(
    localStorage.getItem('search') || ''
  );

  React.useEffect(() => {
    localStorage.setItem('search', searchTerm);
  }, [searchTerm]);
};

const App = () => {
  ...
};
```

So far, this custom hook is just a function around the `useState` and `useEffect` hooks that we've previously used in the `App` component. What's missing is providing an initial state and returning the values that are needed in our `App` component as an array:

src/App.js

```
const useSemiPersistentState = (initialState) => {
  const [searchTerm, setSearchTerm] = React.useState(
    localStorage.getItem('search') || initialState
  );

  React.useEffect(() => {
    localStorage.setItem('search', searchTerm);
  }, [searchTerm]);

  return [searchTerm, setSearchTerm];
};
```

We are following two conventions of React's built-in hooks here. First, the naming convention which puts the "use" prefix in front of every hook name; second, the returned values are returned as an array. Another goal of a custom hook should be reusability. All of this custom hook's internals are about a value of a certain search domain, but the hook should be for a generic value. Let's refactor the naming, therefore:

src/App.js

```
const useSemiPersistentState = (initialState) => {
  const [value, setValue] = React.useState(
    localStorage.getItem('value') || initialState
  );

  React.useEffect(() => {
    localStorage.setItem('value', value);
  }, [value]);

  return [value, setValue];
};
```

We handle an abstracted “value” within the custom hook. Using it in the App component, we can name the returned current state and state updater function anything domain-related (e.g. `searchTerm` and `setSearchTerm`) with array destructuring. There is still one problem with this custom hook. Using the custom hook more than once in a React application leads to an overwrite of the “value”-allocated item in the local storage, because it uses the same key in the local storage. To fix this, pass in an flexible key. Since the key comes from outside, the custom hook assumes that it could change, so it needs to be included in the dependency array of the `useEffect` hook as well. Without it, the side-effect may run with an outdated key (also called *stale*) if the key changed between renders:

src/App.js

```
const useSemiPersistentState = (key, initialState) => {
  const [value, setValue] = React.useState(
    localStorage.getItem(key) || initialState
  );

  React.useEffect(() => {
    localStorage.setItem(key, value);
  }, [value, key]);

  return [value, setValue];
};

const App = () => {
  ...

  const [searchTerm, setSearchTerm] = useSemiPersistentState(
    'search',
    'React'
  );
};
```

```
...  
};
```

You've just created your first custom hook. If you're not comfortable with custom hooks, you can revert the changes and use the `useState` and `useEffect` hook as before, in the `App` component. However, knowing more about custom hooks gives you lots of new options. A custom hook can encapsulate non-trivial implementation details that should be kept away from a component; can be used in more than one React component; can be a composition of other hooks, and can even be open-sourced as an external library. Using your favorite search engine, you'll notice there are hundreds of React hooks that could be used in your application without worry over implementation details.

Exercises:

- Confirm your [source code](#)¹⁰⁸.
 - Confirm the [changes](#)¹⁰⁹.
- Read more about [React Hooks](#)¹¹⁰ to get a good understanding of them, because they are the bread and butter in React function components.
- Optional: [Rate this section](#)¹¹¹.

¹⁰⁸<https://bit.ly/30Koneb>

¹⁰⁹<https://bit.ly/2ZbkAGm>

¹¹⁰<https://www.robinwieruch.de/react-hooks>

¹¹¹<https://forms.gle/5seN1Rv3ZwXmWmDR9>

React Fragments

One caveat with JSX, especially when we create a dedicated Search component, is that we must introduce a wrapping HTML element to render it:

src/App.js

```
const Search = ({ search, onSearch }) => (  
  <div>  
    <label htmlFor="search">Search: </label>  
    <input  
      id="search"  
      type="text"  
      value={search}  
      onChange={onSearch}  
    />  
  </div>  
)
```

Normally the JSX returned by a React component needs only one wrapping top-level element. To render multiple top-level elements side-by-side, we have to wrap them into an array instead. Since we're working with a list of elements, we would have to give every sibling element React's key attribute:

src/App.js

```
const Search = ({ search, onSearch }) => [  
  <label key="1" htmlFor="search">  
    Search: { ' ' }  
  </label>,  
  <input  
    key="2"  
    id="search"  
    type="text"  
    value={search}  
    onChange={onSearch}  
  />,  
]
```

This is one way to have multiple top-level elements in your JSX. It doesn't turn out very readable, though, as it becomes verbose with the additional key attribute. Another solution is to use a **React fragment**:

src/App.js

```
const Search = ({ search, onSearch }) => (  
  <>  
    <label htmlFor="search">Search: </label>  
    <input  
      id="search"  
      type="text"  
      value={search}  
      onChange={onSearch}  
    />  
  </>  
)  
);
```

A fragment wraps other elements into a single top-level element without adding to the rendered output. As an alternative, you can also use `<React.Fragment></React.Fragment>` instead of the shorthand `<></>`. Both Search elements, input field and label, should be visible in your browser now. So if you prefer to omit the wrapping `<div>` or `` elements, substitute them with an empty tag that is allowed in JSX, and doesn't introduce intermediate elements in your rendered HTML.

Exercises:

- Confirm your [source code](#)¹¹².
 - Confirm the [changes](#)¹¹³.
- Read more about [React fragments](#)¹¹⁴.
- Optional: [Rate this section](#)¹¹⁵.

¹¹²<https://bit.ly/3piEnhG>

¹¹³<https://bit.ly/3n9Rmjd>

¹¹⁴<https://bit.ly/2Z4mo44>

¹¹⁵<https://forms.gle/kNpEySPZzckNe6f96>

Reusable React Component

Have a closer look at the Search component. The label element has the text “Search: “; the `id/htmlFor` attributes have the `search` identifier; the value is called `search`, and the callback handler is called `onSearch`. The component is very much tied to the search feature, which makes it less reusable for the rest of the application and non-search-related tasks which would need the same label and input field. Also, it risks introducing bugs if two of these Search components are rendered side by side, because the `htmlFor/id` combination is duplicated, breaking the focus when one of the labels is clicked by the user. Let’s fix these underlying issues by making the Search component reusable.

Since the Search component doesn’t have any actual “search” functionality, it takes little effort to generalize the search domain-specific properties to make the component reusable for the rest of the application. Let’s pass a dynamic `id` and `label` prop to the Search component, rename the actual value and callback handler to something more generic, and rename the component accordingly:

`src/App.js`

```
const App = () => {
  ...

  return (
    <div>
      <h1>My Hacker Stories</h1>

      <InputWithLabel
        id="search"
        label="Search"
        value={searchTerm}
        onChange={handleSearch}
      />

      ...
    </div>
  );
};

const InputWithLabel = ({ id, label, value, onChange }) => (
  <>
    <label htmlFor={id}>{label}</label>
    &nbsp;
    <input
      id={id}
      type="text"
      value={value}
    />
  </>
);
```

```
        onChange={onInputChange}
      />
    </>
  );
```

It's not fully reusable yet. If we want an input field for data like a number (`number`) or phone number (`tel`), the `type` attribute of the input field needs to be accessible from the outside too:

`src/App.js`

```
const InputWithLabel = ({
  id,
  label,
  value,
  type = 'text',
  onInputChange,
}) => (
  <>
    <label htmlFor={id}>{label}</label>
    &nbsp;
    <input
      id={id}
      type={type}
      value={value}
      onChange={onInputChange}
    />
  </>
);
```

Because we don't pass a `type` prop from the `App` component to the `InputWithLabel` component, the [default parameter](#)¹¹⁶ from the function signature takes over for the `type`. Thus, every time the `InputWithLabel` component is used without a `type` prop, the default type will be `text`.

In conclusion, with just a few changes we turned a specialized `Search` component into a more reusable component. We generalized the naming of the internal implementation details and gave the new component a larger API surface to provide all the necessary information from the outside. We aren't using the component elsewhere, but we increased its ability to handle the task if we do.

Exercises:

- Confirm your [source code](#)¹¹⁷.

¹¹⁶<https://mzl.la/3aUefkN>

¹¹⁷<https://bit.ly/3B0roTU>

- Confirm the [changes](#)¹¹⁸.
- Read more about [Reusable React Components](#)¹¹⁹.
- Before we used the text “Search:” with a “:”. How would you deal with it now? Would you pass it with `label="Search:"` as prop to the `InputWithLabel` component or hardcode it after the `<label htmlFor={id}>{label}</label>` usage in the `InputWithLabel` component? We will see how to cope with this later.
- Optional: [Rate this section](#)¹²⁰.

¹¹⁸<https://bit.ly/3C2DzAY>

¹¹⁹<https://www.robinwieruch.de/react-reusable-components>

¹²⁰<https://forms.gle/76C3LvW3kHHwdhgq5>

React Component Composition

The concept of component composition is one of React's more powerful features. Essentially we'll discover how to use a React element in the same fashion as an HTML element, with an opening and closing tag:

src/App.js

```
const App = () => {  
  ...  
  
  return (  
    <div>  
      <h1>My Hacker Stories</h1>  
  
      <InputWithLabel  
        id="search"  
        value={searchTerm}  
        onChange={handleSearch}  
      >  
        Search:  
      </InputWithLabel>  
  
      ...  
    </div>  
  );  
};
```

Instead of using the `label` prop from before, we inserted the text “Search:” between the component's element's tags. In the `InputWithLabel` component, you have access to this information via **React's children** prop now. Instead of using the `label` prop, use the `children` prop to render everything that has been passed down from above where you want it:

src/App.js

```
const InputWithLabel = ({  
  id,  
  value,  
  type = 'text',  
  onChange,  
  children,  
}) => (  
  <>  
    <label htmlFor={id}>{children}</label>
```


Exercises:

- Confirm your [source code](#)¹²¹.
 - Confirm the [changes](#)¹²².
- Read more about [Component Composition in React](#)¹²³.
- Optional: [Rate this section](#)¹²⁴.

¹²¹<https://bit.ly/3lUZqVA>

¹²²<https://bit.ly/3BU66j3>

¹²³<https://www.robinwieruch.de/react-component-composition>

¹²⁴<https://forms.gle/L2GgfHVjAAwbqudq8>

Imperative React

React is inherently declarative, starting with JSX and ending with hooks. In JSX, we tell React *what* to render and not *how* to render it. In a React side-effect Hook (useEffect), we express when to achieve *what* instead of *how* to achieve it. Sometimes, however, we'll want to access the rendered elements of JSX imperatively, most often as a side-effect, in cases such as these:

- read/write access to elements via the DOM API:
 - measuring (read) an element's width or height
 - setting (write) an input field's focus state
- implementation of more complex animations:
 - setting transitions
 - orchestrating transitions
- integration of third-party libraries:
 - [D3¹²⁵](https://d3js.org) is a popular imperative chart library

Because imperative programming in React is often verbose and counterintuitive, we'll walk through only a small example for setting the focus of an input field imperatively. For the declarative way, simply set the input field's autofocus attribute:

src/App.js

```
const InputWithLabel = ({ ... }) => (  
  <>  
    <label htmlFor={id}>{children}</label>  
    &nbsp; <input  
      id={id}  
      type={type}  
      value={value}  
      autoFocus  
      onChange={onInputChange}  
    />  
  </>  
);
```

This works, but only if one of the reusable components is rendered once. For example, if the App component renders two InputWithLabel components, only the last rendered component receives the autofocus on its render. However, since we have a reusable React component here, we can pass a dedicated prop and let the developer decide whether its input field should have autofocus or not:

¹²⁵<https://d3js.org>

src/App.js

```
const App = () => {
  ...

  return (
    <div>
      <h1>My Hacker Stories</h1>

      <InputWithLabel
        id="search"
        value={searchTerm}
        isFocused
        onChange={handleSearch}
      >
        <strong>Search:</strong>
      </InputWithLabel>

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

Using just `isFocused` as an attribute is equivalent to `isFocused={true}`. Within the component, use the new prop for the input field's `autoFocus` attribute:

src/App.js

```
const InputWithLabel = ({
  id,
  value,
  type = 'text',
  onChange,
  isFocused,
  children,
}) => (
  <>
    <label htmlFor={id}>{children}</label>
    &nbsp;
    <input
      id={id}
      type={type}
      value={value}
      autoFocus={isFocused}
    >
```

```
        onChange={onInputChange}
      />
    </>
  );
```

The feature works, yet it's still a declarative implementation. We are telling React *what* to do and not *how* to do it. Even though it's possible to do it with the declarative approach, let's refactor this scenario to an imperative approach. We want to execute the `focus()` method programmatically via the input field element's DOM API once it has been rendered:

src/App.js

```
const InputWithLabel = ({
  id,
  value,
  type = 'text',
  onInputChange,
  isFocused,
  children,
}) => {
  // A
  const inputRef = React.useRef();

  // C
  React.useEffect(() => {
    if (isFocused && inputRef.current) {
      // D
      inputRef.current.focus();
    }
  }, [isFocused]);

  return (
    <>
      <label htmlFor={id}>{children}</label>
      &nbsp;
      { /* B */ }
      <input
        ref={inputRef}
        id={id}
        type={type}
        value={value}
        onChange={onInputChange}
      />
    </>
  );
```

```
);  
};
```

All the essential steps are marked with comments that are explained step by step:

- (A) First, create a `ref` with **React's useRef Hook**. This `ref` object is a persistent value which stays intact over the lifetime of a React component. It comes with a property called `current`, which, in contrast to the `ref` object, can be changed.
- (B) Second, the `ref` is passed to the input field's JSX-reserved `ref` attribute and the element instance is assigned to the changeable `current` property.
- (C) Third, opt into React's lifecycle with React's `useEffect` Hook, performing the focus on the input field when the component renders (or its dependencies change).
- (D) And fourth, since the `ref` is passed to the input field's `ref` attribute, its `current` property gives access to the element. Execute its focus programmatically as a side-effect, but only if `isFocused` is set and the `current` property is existent.

This was an example of how to move from declarative to imperative programming in React. It's not always possible to go the declarative way, so the imperative approach can be performed whenever it's necessary. However, if you can, you should definitely take the declarative approach.

Exercises:

- Confirm your [source code](#)¹²⁶.
 - Confirm the [changes](#)¹²⁷.
- Read more about [refs in React](#)¹²⁸.
- Learn [how to build a Slider component with imperative React](#)¹²⁹.
- Optional: [Rate this section](#)¹³⁰.

¹²⁶<https://bit.ly/3B0qc2S>

¹²⁷<https://bit.ly/3aT3bEq>

¹²⁸<https://www.robinwieruch.de/react-ref>

¹²⁹<https://www.robinwieruch.de/react-slider>

¹³⁰<https://forms.gle/nABoW2tKAPd1yVkv7>

Inline Handler in JSX

The list of stories we have in our App component is only an unstateful variable. We can filter the rendered list with the search feature, but the list itself stays intact even if we apply or remove the filter. The filtered list is just a derived state through a third party (here `searchTerm`), but we can't manipulate the real list yet. To gain control over the list, make it stateful by using it as initial state in React's `useState` Hook. The returned values from the array are the current state (`stories`) and the state updater function (`setStories`):

`src/App.js`

```
const initialStories = [
  {
    title: 'React',
    ...
  },
  {
    title: 'Redux',
    ...
  },
];

...

const App = () => {
  const [searchTerm, setSearchTerm] = ...

  const [stories, setStories] = React.useState(initialStories);

  ...
};
```

The application behaves the same because the `stories`, now returned as a stateful list from React's `useState` Hook, are still filtered into `searchedStories` and displayed in the `List` component. It's just the origin where the stories are coming from have changed. Next, we'll manipulate the list by removing an item from it:

src/App.js

```
const App = () => {
  ...

  const [stories, setStories] = React.useState(initialStories);

  const handleRemoveStory = (item) => {
    const newStories = stories.filter(
      (story) => item.objectID !== story.objectID
    );

    setStories(newStories);
  };

  ...

  return (
    <div>
      <h1>My Hacker Stories</h1>

      ...

      <hr />

      <List list={searchedStories} onRemoveItem={handleRemoveStory} />
    </div>
  );
};
```

The callback handler in the App component – which will be used in the List/Item components eventually – receives the item as an argument which should be removed from the list. Based on this information, the function filters the current stories by removing all items that don't meet its condition. The returned stories – where the desired item (story) has been removed – are then set as a new state and passed to the List component. Since a new state is set, the App component and all components below (e.g. List/Item components) will render again and thus display the new state of stories.

However, what's missing is how the List and Item components are using this new functionality which modifies the state in the App component. The List component itself does not use this new callback handler, but only passes it on to the Item component:

src/App.js

```
const List = ({ list, onRemoveItem }) => (  
  <ul>  
    {list.map((item) => (  
      <Item  
        key={item.objectID}  
        item={item}  
        onRemoveItem={onRemoveItem}  
      />  
    ))}  
  </ul>  
);
```

Finally, the Item component uses the incoming callback handler as a function in a new handler. In this handler, we will pass the specific item to it. Moreover, an additional button element is needed to trigger the actual event:

src/App.js

```
const Item = ({ item, onRemoveItem }) => {  
  const handleRemoveItem = () => {  
    onRemoveItem(item);  
  };  
  
  return (  
    <li>  
      <span>  
        <a href={item.url}>{item.title}</a>  
      </span>  
      <span>{item.author}</span>  
      <span>{item.num_comments}</span>  
      <span>{item.points}</span>  
      <span>  
        <button type="button" onClick={handleRemoveItem}>  
          Dismiss  
        </button>  
      </span>  
    </li>  
  );  
};
```

So far in this section, we have made the list of stories stateful with React's useState Hook; passed the still searched stories down as props to the List component, and implemented a callback handler

(`handleRemoveStory`) and handler (`handleRemoveItem`) to be used in their respective components to remove a story by clicking on a button. In order to implement this feature, we applied many lessons learned from before: state, props, handlers, and callback handlers.

You may have noticed that we had to introduce an additional `handleRemoveItem` handler in the `Item` component which is in charge to execute the incoming `onRemoveItem` callback handler. If you want to make this more elegant, you can use an **inline handler** which would allow you to execute the callback handler function in the `Item` component right in the JSX. There are two solutions using the incoming `onRemoveItem` function in the `Item` component as an inline handler. First, using JavaScript's `bind` method:

src/App.js

```
const Item = ({ item, onRemoveItem }) => (  
  <li>  
    <span>  
      <a href={item.url}>{item.title}</a>  
    </span>  
    <span>{item.author}</span>  
    <span>{item.num_comments}</span>  
    <span>{item.points}</span>  
    <span>  
      <button type="button" onClick={onRemoveItem.bind(null, item)}>  
        Dismiss  
      </button>  
    </span>  
  </li>  
>);
```

Using [JavaScript's `bind` method](https://mzl.la/3ncEkBu)¹³¹ on a function allows us to bind arguments directly to that function that should be used when executing it. The `bind` method returns a new function with the bound argument attached. In contrast, the second and more popular solution is to use a wrapping arrow function, which allows us to sneak in arguments like `item`:

¹³¹<https://mzl.la/3ncEkBu>

src/App.js

```
const Item = ({ item, onRemoveItem }) => (  
  <li>  
    <span>  
      <a href={item.url}>{item.title}</a>  
    </span>  
    <span>{item.author}</span>  
    <span>{item.num_comments}</span>  
    <span>{item.points}</span>  
    <span>  
      <button type="button" onClick={() => onRemoveItem(item)}>  
        Dismiss  
      </button>  
    </span>  
  </li>  
>);
```

This is a quick solution, because sometimes we don't want to refactor a function component's concise function body back to a block body to define an appropriate handler between function signature and return statement. While this way is more concise than the others, it can also be more difficult to debug because JavaScript logic may be hidden in JSX. It becomes even more verbose if the wrapping arrow function encapsulates more than one line of implementation logic, by using a block body instead of a concise body. This should be avoided:

Code Playground

```
const Item = ({ item, onRemoveItem }) => (  
  <li>  
    ...  
    <span>  
      <button  
        type="button"  
        onClick={() => {  
          // do something else  
  
          // note: avoid using complex logic in JSX  
  
          onRemoveItem(item);  
        }}  
      >  
        Dismiss  
      </button>  
    </span>
```

```
</li>  
);
```

All three handler versions, two of which are inline and the normal handler, are acceptable. The non-inlined handler moves the implementation details into the function component's block body; both inline handler versions move the implementation details into the JSX.

Exercises:

- Confirm your [source code](#)¹³².
 - Confirm the [changes](#)¹³³.
- Read more about how to [add](#)¹³⁴, [update](#)¹³⁵, [remove](#)¹³⁶ items in a list.
- Read more about [computed properties in React](#)¹³⁷.
- Review handlers, callback handlers, and inline handlers.
- Optional: [Rate this section](#)¹³⁸.

¹³²<https://bit.ly/3vrGWzb>

¹³³<https://bit.ly/3jj37CR>

¹³⁴<https://www.robinwieruch.de/react-add-item-to-list>

¹³⁵<https://www.robinwieruch.de/react-update-item-in-list>

¹³⁶<https://www.robinwieruch.de/react-remove-item-from-list>

¹³⁷<https://www.robinwieruch.de/react-computed-properties>

¹³⁸<https://forms.gle/19NvNYMk2RUKTDyZ6>

React Asynchronous Data

We have two interactions in our application: searching the list and removing items from the list. The first interaction is a fluctuant interference through a third-party state (`searchTerm`) applied on the list; the second interaction is a non-reversible deletion of an item from the list. However, the list we are dealing with is still just sample data. What about preparing our application to deal with real data instead?

Usually, data from a remote backend/database arrives asynchronously for client-side applications like React. Thus it's often the case that we must render a component before we can initiate the data fetching. In the following, we will start by simulating this kind of asynchronous data with our sample data in the application. Later, we will replace it with real data fetched from a real remote API. We start off with a function that returns a promise with data in its shorthand version once it resolves. The resolved object holds the previous list of stories:

src/App.js

```
const initialStories = [ ... ];

const getAsyncStories = () =>
  Promise.resolve({ data: { stories: initialStories } });
```

In the App component, instead of using the `initialStories`, use an empty array for the initial state. We want to start off with an empty list of stories and simulate fetching these stories asynchronously. In a new `useEffect` hook, call the function and resolve the returned promise as a side-effect. Due to the empty dependency array, the side-effect only runs once the component renders for the first time:

src/App.js

```
const App = () => {
  ...

  const [stories, setStories] = React.useState([]);

  React.useEffect(() => {
    getAsyncStories().then(result => {
      setStories(result.data.stories);
    });
  }, []);

  ...
};
```

Even though the data should arrive asynchronously when we start the application, it appears to arrive synchronously, because it's rendered immediately. Let's change this by giving it a bit of a realistic delay, because every network request to a remote API would come with a delay. First, remove the shorthand version for the promise:

src/App.js

```
const getAsyncStories = () =>
  new Promise((resolve) =>
    resolve({ data: { stories: initialStories } })
  );
```

And second, when resolving the promise, delay it for a few seconds:

src/App.js

```
const getAsyncStories = () =>
  new Promise((resolve) =>
    setTimeout(
      () => resolve({ data: { stories: initialStories } }),
      2000
    )
  );
```

Once you start the application again, you should see a delayed rendering of the list. The initial state for the stories is an empty array. After the App component is rendered, the side-effect hook runs once to fetch the asynchronous data. After resolving the promise and setting the data in the component's state, the component renders again and displays the list of asynchronously loaded stories.

Exercises:

- Confirm your [source code](#)¹³⁹.
 - Confirm the [changes](#)¹⁴⁰.
- Read more about [JavaScript Promises](#)¹⁴¹.
- Read more about [faking a remote API with JavaScript](#)¹⁴².
- Read more about [using mock data in React](#)¹⁴³.
- Optional: [Rate this section](#)¹⁴⁴.

¹³⁹<https://bit.ly/3vu6kEb>

¹⁴⁰<https://bit.ly/3B0p7rQ>

¹⁴¹<https://mzl.la/3aTGuzQz>

¹⁴²<https://www.robinwieruch.de/javascript-fake-api>

¹⁴³<https://www.robinwieruch.de/react-mock-data>

¹⁴⁴<https://forms.gle/sfQcc477xmgGRlyB7>

React Conditional Rendering

A **conditional rendering** in React always happens if we have to render different JSX based on state or props. Dealing with asynchronous data is a good use case for dealing with these conditional states. For example, when the application initializes for the first time, there is no data to start with. Next, we are loading data and eventually, we have the data at our disposal to display it. Sometimes the data fetching fails and we receive an error instead. So there are lots of things to cover for us as developers.

Fortunately, a few of these cases are already taken care of. For instance, because our initial state is an empty list rather than `null`, we don't have to worry that this breaks the application when we filter and map over this list. However, some things are still missing. For example, let's introduce one more state for a loading indicator which gives our users feedback about the pending data request:

src/App.js

```
const App = () => {  
  ...  
  
  const [stories, setStories] = React.useState([]);  
  const [isLoading, setIsLoading] = React.useState(false);  
  
  React.useEffect(() => {  
    setIsLoading(true);  
  
    getAsyncStories().then((result) => {  
      setStories(result.data.stories);  
      setIsLoading(false);  
    });  
  }, []);  
  
  ...  
};
```

With [JavaScript's ternary operator](#)¹⁴⁵, we can inline this conditional state as a **conditional rendering** in JSX:

¹⁴⁵<https://mzl.la/3vAPKCL>

src/App.js

```
const App = () => {
  ...

  return (
    <div>
      ...

      <hr />

      {isLoading ? (
        <p>Loading ...</p>
      ) : (
        <List
          list={searchedStories}
          onRemoveItem={handleRemoveStory}
        />
      )}
    </div>
  );
};
```

Asynchronous data comes with error handling, too. It doesn't happen in our simulated environment, but there could be errors if we start fetching data from another third-party API. Introduce another state for error handling and handle it in the promise's `catch()` block when resolving the promise:

src/App.js

```
const App = () => {
  ...

  const [stories, setStories] = React.useState([]);
  const [isLoading, setIsLoading] = React.useState(false);
  const [isError, setIsError] = React.useState(false);

  React.useEffect(() => {
    setIsLoading(true);

    getAsyncStories()
      .then((result) => {
        setStories(result.data.stories);
        setIsLoading(false);
      })
  });
};
```

```
    .catch(() => setIsError(true));
  }, []);

  ...
};
```

Next, give the user feedback in case something goes wrong with another conditional rendering. This time, it's either rendering something or nothing. So instead of having a ternary operator where one side returns `null`, use the logical `&&` operator as shorthand:

src/App.js

```
const App = () => {
  ...

  return (
    <div>
      ...

      <hr />

      {isError && <p>Something went wrong ...</p>}

      {isLoading ? (
        <p>Loading ...</p>
      ) : (
        ...
      )}
    </div>
  );
};
```

In JavaScript, a `true && 'Hello World'` always evaluates to `'Hello World'`. A `false && 'Hello World'` always evaluates to `false`. In React, we can use this behaviour to our advantage. If the condition is true, the expression after the logical `&&` operator will be the output. If the condition is false, React ignores it and skips the expression.

Conditional rendering is not just for asynchronous data though. The simplest example of conditional rendering is a boolean flag state that's toggled with a button. If the boolean flag is true, render something, if it is false, don't render anything. Knowing about this feature in React can be quite powerful, because it gives you the ability to conditionally render JSX. It's yet another tool in React to make your UI more dynamic. And as we've discovered, it's often necessary for more complex control flows like asynchronous data.

Exercises:

- Confirm your [source code](#)¹⁴⁶.
 - Confirm the [changes](#)¹⁴⁷.
- Read more about [conditional rendering in React](#)¹⁴⁸.
- Optional: [Rate this section](#)¹⁴⁹.

¹⁴⁶<https://bit.ly/2ZfiJLM>

¹⁴⁷<https://bit.ly/3AYmneE>

¹⁴⁸<https://www.robinwieruch.de/conditional-rendering-react/>

¹⁴⁹<https://forms.gle/kHLAXtMaKsTFtWjY9>

React Advanced State

All state management in this application makes heavy use of React's `useState` Hook. On the other hand, React's **`useReducer` Hook** gives you more sophisticated state management. Since the concept of reducers in JavaScript splits the community in half, we won't cover it extensively here, but the exercises at the end of this section should give you plenty of practice.

We'll move the `stories` state management from the `useState` hook to a new `useReducer` hook. First, introduce a reducer function outside of your components. A reducer function always receives `state` and `action`. Based on these two arguments, a reducer always returns a new state:

src/App.js

```
const storiesReducer = (state, action) => {
  if (action.type === 'SET_STORIES') {
    return action.payload;
  } else {
    throw new Error();
  }
};
```

A reducer `action` is always associated with a `type`. If this type matches a condition in the reducer, do something. If it isn't covered by the reducer, throw an error to remind yourself the implementation isn't covered. The `storiesReducer` function covers one `type` and then returns the `payload` of the incoming action without using the current state to compute the new state. The new state is simply the `payload`.

In the `App` component, exchange `useState` for `useReducer` for managing the `stories`. The new hook receives a reducer function and an initial state as arguments and returns an array with two items. The first item is the *current state*; the second item is the *state updater function* (also called *dispatch function*):

src/App.js

```
const App = () => {
  ...

  const [stories, dispatchStories] = React.useReducer(
    storiesReducer,
    []
  );

  ...
};
```

The new `dispatch` function can be used instead of the `setStories` function, which was previously returned from `useState`. Instead of setting the state explicitly with the state updater function from `useState`, the `useReducer` state updater function dispatches an action for the reducer. The action comes with a type and an optional payload:

src/App.js

```
const App = () => {
  ...

  React.useEffect(() => {
    setIsLoading(true);

    getAsyncStories()
      .then((result) => {
        dispatchStories({
          type: 'SET_STORIES',
          payload: result.data.stories,
        });
        setIsLoading(false);
      })
      .catch(() => setIsError(true));
  }, []);

  const handleRemoveStory = (item) => {
    const newStories = stories.filter(
      (story) => item.objectID !== story.objectID
    );

    dispatchStories({
      type: 'SET_STORIES',
      payload: newStories,
    });
  };

  ...
};
```

The application appears the same in the browser, though a reducer and React's `useReducer` hook are managing the state for the stories now. Let's bring the concept of a reducer to a minimal version by handling more than one state transition. So far, the `handleRemoveStory` handler computes the new stories. It's valid to move this logic into the reducer function and manage the reducer with an action, which is another case for moving from imperative to declarative programming. Instead of doing it

ourselves by saying *how it should be done*, we are telling the reducer *what to do*. Everything else is hidden in the reducer:

src/App.js

```
const App = () => {  
  ...  
  
  const handleRemoveStory = (item) => {  
    dispatchStories({  
      type: 'REMOVE_STORY',  
      payload: item,  
    });  
  };  
  
  ...  
};
```

Now the reducer function has to cover this new case in a new conditional state transition. If the condition for removing a story is met, the reducer has all the implementation details needed to remove the story. The action gives all the necessary information, an item's identifier, to remove the story from the current state and return a new list of filtered stories as state:

src/App.js

```
const storiesReducer = (state, action) => {  
  if (action.type === 'SET_STORIES') {  
    return action.payload;  
  } else if (action.type === 'REMOVE_STORY') {  
    return state.filter(  
      (story) => action.payload.objectID !== story.objectID  
    );  
  } else {  
    throw new Error();  
  }  
};
```

All these if-else statements will eventually clutter when adding more state transitions into one reducer function. Refactoring it to a switch statement for all the state transitions makes it more readable:

src/App.js

```
const storiesReducer = (state, action) => {
  switch (action.type) {
    case 'SET_STORIES':
      return action.payload;
    case 'REMOVE_STORY':
      return state.filter(
        (story) => action.payload.objectID !== story.objectID
      );
    default:
      throw new Error();
  }
};
```

What we've covered is a minimal version of a reducer in JavaScript. It covers two state transitions, shows how to compute the current state and action into a new state, and uses some business logic (removal of a story). Now we can set a list of stories as state for the asynchronously arriving data, and remove a story from the list of stories, with just one state managing reducer and its associated `useReducer` hook. To fully grasp the concept of reducers in JavaScript and the usage of React's `useReducer` Hook, visit the linked resources in the exercises.

Exercises:

- Confirm your [source code](#)¹⁵⁰.
 - Confirm the [changes](#)¹⁵¹.
- Read more about [reducers in JavaScript](#)¹⁵².
- Read more about [reducers and useReducer in React](#)¹⁵³.
- Extract the action types 'SET_STORIES' and 'REMOVE_STORY' as variables and reuse them in the reducer and the dispatch functions. This way, you will avoid introducing typos in your action types.
- Optional: [Rate this section](#)¹⁵⁴.

¹⁵⁰<https://bit.ly/3nbb5Pp>

¹⁵¹<https://bit.ly/3lZrV4y>

¹⁵²<https://www.robinwieruch.de/javascript-reducer>

¹⁵³<https://www.robinwieruch.de/react-usereducer-hook>

¹⁵⁴<https://forms.gle/tNqQVynwQV9Ym9u68>

React Impossible States

Perhaps you've noticed a disconnect between the single states in the App component, which seem to belong together because of the `useState` hooks. Technically, all the states related to the asynchronous data belong together, which doesn't only include the stories as actual data, but also their loading and error states.

There is nothing wrong with multiple `useState` hooks in one React component. Be wary once you see multiple state updater functions in a row, however. These conditional states can lead to **impossible states** and undesired behavior in the UI. Try changing your pseudo data fetching function to the following to simulate the error handling:

src/App.js

```
const getAsyncStories = () =>
  new Promise((resolve, reject) => setTimeout(reject, 2000));
```

The impossible state happens when an error occurs for the asynchronous data. The state for the error is set, but the state for the loading indicator isn't revoked. In the UI, this would lead to an infinite loading indicator and an error message, though it may be better to show the error message only and hide the loading indicator. Impossible states are not easy to spot, which makes them infamous for causing bugs in the UI.

Fortunately, we can improve our chances by moving states that belong together from multiple `useState` and `useReducer` hooks into a single `useReducer` hook. Take the following `useState` hooks:

src/App.js

```
const App = () => {
  ...

  const [stories, dispatchStories] = React.useReducer(
    storiesReducer,
    []
  );
  const [isLoading, setIsLoading] = React.useState(false);
  const [isError, setIsError] = React.useState(false);

  ...
};
```

And merge them into one `useReducer` hook for a unified state management and a more complex state object:

src/App.js

```
const App = () => {  
  ...  
  
  const [stories, dispatchStories] = React.useReducer(  
    storiesReducer,  
    { data: [], isLoading: false, isError: false }  
  );  
  
  ...  
};
```

Now everything related to asynchronous data fetching must use the new dispatch function for state transitions:

src/App.js

```
const App = () => {  
  ...  
  
  const [stories, dispatchStories] = React.useReducer(  
    storiesReducer,  
    { data: [], isLoading: false, isError: false }  
  );  
  
  React.useEffect(() => {  
    dispatchStories({ type: 'STORIES_FETCH_INIT' });  
  
    getAsyncStories()  
      .then((result) => {  
        dispatchStories({  
          type: 'STORIES_FETCH_SUCCESS',  
          payload: result.data.stories,  
        });  
      })  
      .catch(() =>  
        dispatchStories({ type: 'STORIES_FETCH_FAILURE' })  
      );  
  }, []);  
  
  ...  
};
```

Since we introduced new types for state transitions and a new state structure, we must add these types and change the structure in the `storiesReducer` reducer function:

`src/App.js`

```
const storiesReducer = (state, action) => {
  switch (action.type) {
    case 'STORIES_FETCH_INIT':
      return {
        ...state,
        isLoading: true,
        isError: false,
      };
    case 'STORIES_FETCH_SUCCESS':
      return {
        ...state,
        isLoading: false,
        isError: false,
        data: action.payload,
      };
    case 'STORIES_FETCH_FAILURE':
      return {
        ...state,
        isLoading: false,
        isError: true,
      };
    case 'REMOVE_STORY':
      return {
        ...state,
        data: state.data.filter(
          (story) => action.payload.objectID !== story.objectID
        ),
      };
    default:
      throw new Error();
  }
};
```

For every state transition, we return a *new state* object which contains all the key/value pairs from the *current state* object (via JavaScript's spread operator) and the new overwriting properties. For example, `STORIES_FETCH_FAILURE` resets the `isLoading`, sets the `isError` boolean flags, while keeping all the other state intact (e.g. `stories`). That's how we get around the bug introduced earlier as impossible state since an error should remove the loading state.

Observe how the `REMOVE_STORY` action changed as well. It operates on the `state.data`, and no longer just on the plain state. The state is a complex object with `data`, `loading`, and `error` states rather than just a list of stories. This has to be solved in the remaining code too:

`src/App.js`

```
const App = () => {
  ...

  const [stories, dispatchStories] = React.useReducer(
    storiesReducer,
    { data: [], isLoading: false, isError: false }
  );

  ...

  const searchedStories = stories.data.filter((story) =>
    story.title.toLowerCase().includes(searchTerm.toLowerCase())
  );

  return (
    <div>
      ...

      {stories.isError && <p>Something went wrong ...</p>}

      {stories.isLoading ? (
        <p>Loading ...</p>
      ) : (
        <List
          list={searchedStories}
          onRemoveItem={handleRemoveStory}
        />
      )}
    </div>
  );
};
```

Try to use the erroneous data fetching function again and check whether everything works as expected now:

src/App.js

```
const getAsyncStories = () =>
  new Promise((resolve, reject) => setTimeout(reject, 2000));
```

We moved from unreliable state transitions with multiple `useState` hooks to predictable state transitions with React's `useReducer` Hook. The state object managed by the reducer encapsulates everything related to the stories, including loading and error state, but also implementation details like removing a story from the list of stories. We didn't get fully rid of impossible states, because it's still possible to leave out a crucial boolean flag like before, but we moved one step closer towards more predictable state management.

Exercises:

- Confirm your [source code](#)¹⁵⁵.
 - Confirm the [changes](#)¹⁵⁶.
- Read over the previously linked tutorials about reducers in JavaScript and React.
- Read more about [when to use `useState` or `useReducer` in React](#)¹⁵⁷.
- Read more about [deriving state from props in React](#)¹⁵⁸.
- Optional: [Rate this section](#)¹⁵⁹.

¹⁵⁵<https://bit.ly/3G6AphY>

¹⁵⁶<https://bit.ly/3jepMA7>

¹⁵⁷<https://www.robinwieruch.de/react-usereducer-vs-usestate>

¹⁵⁸<https://www.robinwieruch.de/react-derive-state-props>

¹⁵⁹<https://forms.gle/XWTJS65iu6WkiZMCA>

Data Fetching with React

We are currently fetching data, but it's still pseudo data coming from a promise we set up ourselves. The lessons up to now about asynchronous React and advanced state management were preparing us to fetch data from a real remote third-party API. We will use the reliable and informative [Hacker News API¹⁶⁰](https://hn.algolia.com/api/v1/search?query=) to request popular tech stories.

Instead of using the `initialStories` array and `getAsyncStories` function (you can remove these), we will fetch the data directly from the API:

`src/App.js`

```
// A
const API_ENDPOINT = 'https://hn.algolia.com/api/v1/search?query=';

const App = () => {
  ...

  React.useEffect(() => {
    dispatchStories({ type: 'STORIES_FETCH_INIT' });

    fetch(`${API_ENDPOINT}react`) // B
      .then((response) => response.json()) // C
      .then((result) => {
        dispatchStories({
          type: 'STORIES_FETCH_SUCCESS',
          payload: result.hits, // D
        });
      })
      .catch(() =>
        dispatchStories({ type: 'STORIES_FETCH_FAILURE' })
      );
  }, []);

  ...
};
```

First, the `API_ENDPOINT` (A) is used to fetch popular tech stories for a certain query (a search term). In this case, we fetch stories about React (B). Second, the native browser's [fetch API¹⁶¹](https://developer.mozilla.org/en-US/docs/Web/API/Window.fetch) is used to make this request (B). For the fetch API, the response needs to be translated into JSON (C). Finally, the returned result follows a different data structure (D), which we send as payload to our component's state reducer.

¹⁶⁰<https://hn.algolia.com/api>

¹⁶¹<https://mzl.la/2Z1kyjU>

In the previous code example, we used [JavaScript's Template Literals](#)¹⁶² for a string interpolation. When this feature wasn't available in JavaScript, we'd have used the + operator on strings instead:

Code Playground

```
const greeting = 'Hello';

// + operator
const welcome = greeting + ' React';
console.log(welcome);
// Hello React

// template literals
const anotherWelcome = `${greeting} React`;
console.log(anotherWelcome);
// Hello React
```

Check your browser to see stories related to the initial query fetched from the Hacker News API. Since we used the same data structure for a story for the sample stories, we didn't need to change anything, and it's still possible to filter the stories after fetching them with the search feature. We will change this behavior in one of the next sections though.

Exercises:

- Confirm your [source code](#)¹⁶³.
 - Confirm the [changes](#)¹⁶⁴.
- Read through [Hacker News](#)¹⁶⁵ and its [API](#)¹⁶⁶.
- Read more about [the browser native fetch API](#)¹⁶⁷ for connecting to remote APIs.
- Read more about [JavaScript's Template Literals](#)¹⁶⁸.
- Optional: [Rate this section](#)¹⁶⁹.

¹⁶²<https://mzl.la/3jlcVfn>

¹⁶³<https://bit.ly/3DYWxZS>

¹⁶⁴<https://bit.ly/3jks1Sj>

¹⁶⁵<https://news.ycombinator.com>

¹⁶⁶<https://hn.algolia.com/api>

¹⁶⁷<https://mzl.la/2Z1kyjU>

¹⁶⁸<https://mzl.la/3jlcVfn>

¹⁶⁹<https://forms.gle/hoJxjpoZQGCS7Vp9>

Data Re-Fetching in React

So far, the App component fetches a list of stories once with a predefined query (react). After that, users can search for stories on the client-side. Now we'll move this feature from client-side to server-side searching, using the actual `searchTerm` as a dynamic query for the API request.

First, remove `searchedStories` because we will receive the stories searched from the API. Pass only the regular stories to the List component:

src/App.js

```
const App = () => {
  ...

  return (
    <div>
      ...

      {stories.isLoading ? (
        <p>Loading ...</p>
      ) : (
        <List list={stories.data} onRemoveItem={handleRemoveStory} />
      )}
    </div>
  );
};
```

And second, instead of using a hardcoded search term like before, use the actual `searchTerm` from the component's state. If `searchTerm` is an empty string, do nothing:

src/App.js

```
const App = () => {
  ...

  React.useEffect(() => {
    if (searchTerm === '') return;

    dispatchStories({ type: 'STORIES_FETCH_INIT' });

    fetch(`${API_ENDPOINT}${searchTerm}`)
      .then((response) => response.json())
      .then((result) => {
        dispatchStories({
```

```

        type: 'STORIES_FETCH_SUCCESS',
        payload: result.hits,
      });
    })
    .catch(() =>
      dispatchStories({ type: 'STORIES_FETCH_FAILURE' })
    );
  }, []);

  ...
};

```

The initial search respects the search term now. If the `searchTerm` changes however, we need to run the side-effect for the data fetching again. In addition, if `searchTerm` is not present (e.g. null, empty string, undefined), do nothing (as a more generalized condition):

src/App.js

```

const App = () => {
  ...

  React.useEffect(() => {
    if (!searchTerm) return;

    dispatchStories({ type: 'STORIES_FETCH_INIT' });

    fetch(`${API_ENDPOINT}${searchTerm}`)
      .then((response) => response.json())
      .then((result) => {
        dispatchStories({
          type: 'STORIES_FETCH_SUCCESS',
          payload: result.hits,
        });
      })
      .catch(() =>
        dispatchStories({ type: 'STORIES_FETCH_FAILURE' })
      );
    }, [searchTerm]);

  ...
};

```

We changed the feature from a client-side to server-side search. Instead of filtering a predefined list of stories on the client, the `searchTerm` is used to fetch a server-side filtered list. The server-side

search happens not only for the initial data fetching, but also for data fetching if the `searchTerm` changes. The feature is fully server-side now. Re-fetching data each time someone types into the input field isn't optimal though, so we'll correct that soon. Because this implementation stresses the API, you might experience errors if you use requests too often.

Exercises:

- Confirm your [source code](#)¹⁷⁰.
 - Confirm the [changes](#)¹⁷¹.
- Optional: [Rate this section](#)¹⁷².

¹⁷⁰<https://bit.ly/3lZtjUO>

¹⁷¹<https://bit.ly/3b9S4aF>

¹⁷²<https://forms.gle/ywE4bFy6D2HSG8Rd7>

Memoized Handler in React (Advanced)

The previous sections have taught you about handlers, callback handlers, and inline handlers. Now we'll introduce a **memoized handler**, which can be applied on top of handlers and callback handlers. For the sake of learning, we will move all the data fetching logic into a standalone function outside the side-effect (A); wrap it into a `useCallback` hook (B), and then invoke it in the `useEffect` hook (C):

src/App.js

```
const App = () => {
  ...

  // A
  const handleFetchStories = React.useCallback(() => { // B
    if (!searchTerm) return;

    dispatchStories({ type: 'STORIES_FETCH_INIT' });

    fetch(`${API_ENDPOINT}${searchTerm}`)
      .then((response) => response.json())
      .then((result) => {
        dispatchStories({
          type: 'STORIES_FETCH_SUCCESS',
          payload: result.hits,
        });
      })
      .catch(() =>
        dispatchStories({ type: 'STORIES_FETCH_FAILURE' })
      );
  }, [searchTerm]); // E

  React.useEffect(() => {
    handleFetchStories(); // C
  }, [handleFetchStories]); // D

  ...
};
```

The application behaves the same, only the implementation logic has been refactored. Instead of using the data fetching logic anonymously in a side-effect, we made it available as a function for the application. Let's explore why React's `useCallback` Hook is needed here. This hook creates a

memoized function every time its dependency array (E) changes. As a result, the `useEffect` hook runs again (C) because it depends on the new function (D):

Visualization

1. change: `searchTerm`
 2. implicit change: `handleFetchStories`
 3. run: side-effect
-

If we didn't create a memoized function with React's `useCallback` Hook, a new `handleFetchStories` function would be created each time the App component re-renders, and would be executed in the `useEffect` hook to fetch data. The fetched data is then stored as state in the component. Because the state of the component changed, the component re-renders and creates a new `handleFetchStories` function. The side-effect would be triggered to fetch data, and we'd be stuck in an endless loop:

Visualization

1. define: `handleFetchStories`
 2. run: side-effect
 3. update: state
 4. re-render: component
 5. re-define: `handleFetchStories`
 6. run: side-effect
 - ...
-

React's `useCallback` hook changes the function only when the search term changes. That's when we want to trigger a re-fetch of the data, because the input field has new input and we want to see the new data displayed in our list.

By moving the data fetching function outside the `useEffect` hook, it becomes reusable for other parts of the application. We won't use it just yet, but it is a way to understand the `useCallback` hook. Now the `useEffect` hook runs implicitly when the `searchTerm` changes, because the `handleFetchStories` is re-defined each time the `searchTerm` changes. Since the `useEffect` hook depends on the `handleFetchStories`, the side-effect for data fetching runs again.

Exercises:

- Confirm your [source code](#)¹⁷³.
 - Confirm the [changes](#)¹⁷⁴.
- Read more about [React's useCallback Hook](#)¹⁷⁵.
- Optional: [Rate this section](#)¹⁷⁶.

¹⁷³<https://bit.ly/3aSpb2v>

¹⁷⁴<https://bit.ly/3G4vkGX>

¹⁷⁵<https://www.robinwieruch.de/react-usecallback-hook>

¹⁷⁶<https://forms.gle/HSX9aurgsf5j76HR9>

Explicit Data Fetching with React

Re-fetching all data each time someone types in the input field isn't optimal. Since we're using a third-party API to fetch the data, its internals is out of our reach. Eventually, we will incur [rate limiting](#)¹⁷⁷, which returns an error instead of data. To solve this problem, change the implementation details from implicit to explicit data (re-)fetching. In other words, the application will refetch data only if someone clicks a confirmation button.

First, add a button element for the confirmation to the JSX:

src/App.js

```
const App = () => {
  ...

  return (
    <div>
      <h1>My Hacker Stories</h1>

      <InputWithLabel
        id="search"
        value={searchTerm}
        isFocused
        onChange={handleSearchInput}
      >
        <strong>Search:</strong>
      </InputWithLabel>

      <button
        type="button"
        disabled={!searchTerm}
        onClick={handleSearchSubmit}
      >
        Submit
      </button>

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

Second, the handler, input, and button handler receive implementation logic to update the component's state. The input field handler still updates the `searchTerm`; the button handler sets the `url`

¹⁷⁷<https://bit.ly/2ZaJXl8>

derived from the *current* `searchTerm` and the static API URL as a new state:

src/App.js

```
const App = () => {
  const [searchTerm, setSearchTerm] = useSemiPersistentState(
    'search',
    'React'
  );

  const [url, setUrl] = React.useState(
    `${API_ENDPOINT}${searchTerm}`
  );

  ...

  const handleSearchInput = (event) => {
    setSearchTerm(event.target.value);
  };

  const handleSearchSubmit = () => {
    setUrl(`${API_ENDPOINT}${searchTerm}`);
  };

  ...
};
```

Third, instead of running the data fetching side-effect on every `searchTerm` change – which would happen each time the input field’s value changes – the `url` is used. The `url` is set explicitly by the user when the search is confirmed via our new button:

src/App.js

```
const App = () => {
  ...

  const handleFetchStories = React.useCallback(() => {
    dispatchStories({ type: 'STORIES_FETCH_INIT' });

    fetch(url)
      .then((response) => response.json())
      .then((result) => {
        dispatchStories({
          type: 'STORIES_FETCH_SUCCESS',
          payload: result.hits,
        });
      });
  });
};
```

```
    });
  })
  .catch(() =>
    dispatchStories({ type: 'STORIES_FETCH_FAILURE' })
  );
}, [ur1]);

React.useEffect(() => {
  handleFetchStories();
}, [handleFetchStories]);

...
};
```

Before the `searchTerm` was used for two cases: updating the input field's state and activating the side-effect for fetching data. One may consider this too many responsibilities for the `searchTerm`. Now it's only used for the former. A second state called `ur1` got introduced for triggering the side-effect for fetching data which only happens when a user clicks the confirmation button.

Exercises:

- Confirm your [source code](#)¹⁷⁸.
 - Confirm the [changes](#)¹⁷⁹.
- Why is `useState` instead of `useSemiPersistentState` used for the `ur1` state management?
- Why is there no check for an empty `searchTerm` in the `handleFetchStories` function anymore?
- Optional: [Rate this section](#)¹⁸⁰.

¹⁷⁸<https://bit.ly/3n47Qcw>

¹⁷⁹<https://bit.ly/3AXZk3J>

¹⁸⁰<https://forms.gle/HuJDuvNZmEDbhGzU9>

Third-Party Libraries in React

We previously introduced the native fetch API to complete requests to the Hacker News API, which the browser provides. However, not all browsers support this, especially the older ones. Also, once you start testing your application in a [headless browser environment¹⁸¹](#), issues can arise with the fetch API. There are a couple of ways to make fetch work in older browsers ([polyfills¹⁸²](#)) and in tests (isomorphic fetch), but these concepts are a bit off-task for the purpose of this learning experience.

One alternative is to substitute the native fetch API with a stable library like [axios¹⁸³](#), which performs asynchronous requests to remote APIs. In this section, we will discover how to substitute a library – a native API of the browser in this case – with another library from the npm registry. First, install axios from the command line:

Command Line

```
npm install axios
```

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

src/App.js

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

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

src/App.js

```
const App = () => {  
  ...  
  
  const handleFetchStories = React.useCallback(() => {  
    dispatchStories({ type: 'STORIES_FETCH_INIT' });  
  
    axios  
    .get(url)  
    .then((result) => {
```

¹⁸¹<https://bit.ly/3ncFfSs>

¹⁸²<https://bit.ly/3ASC86Y>

¹⁸³<https://bit.ly/3jjEupg>

```
    dispatchStories({
      type: 'STORIES_FETCH_SUCCESS',
      payload: result.data.hits,
    });
  })
  .catch(() =>
    dispatchStories({ type: 'STORIES_FETCH_FAILURE' })
  );
}, [url]);

...
};
```

In this code, we call `axios.get()` for an explicit [HTTP GET request¹⁸⁴](#), which is the same HTTP method we used by default with the browser's native `fetch` API. You can use other HTTP methods such as HTTP POST with `axios.post()` as well. We can see with these examples that `axios` is a powerful library for performing requests to remote APIs. I recommend it over the native `fetch` API when requests become complex, working with older browsers, or for testing.

Exercises:

- Confirm your [source code¹⁸⁵](#).
 - Confirm the [changes¹⁸⁶](#).
- Read more about [popular libraries in React¹⁸⁷](#).
- Read more about [why frameworks matter¹⁸⁸](#).
- Read more about [axios¹⁸⁹](#).
- Optional: [Rate this section¹⁹⁰](#).

¹⁸⁴<https://mzl.la/3n5kUyi>

¹⁸⁵<https://bit.ly/3piaFt4>

¹⁸⁶<https://bit.ly/3jjMruf>

¹⁸⁷<https://www.robinwieruch.de/react-libraries>

¹⁸⁸<https://www.robinwieruch.de/why-frameworks-matter>

¹⁸⁹<https://bit.ly/3jjEupg>

¹⁹⁰<https://forms.gle/wfDb7r5K4az3TiWN9>

Async/Await in React (Advanced)

You'll work with asynchronous data often in React, so it's good to know an alternative syntax for handling promises: `async/await`. The following refactoring of the `handleFetchStories` function (without error handling) shows how:

src/App.js

```
const App = () => {  
  ...  
  
  const handleFetchStories = React.useCallback(async () => {  
    dispatchStories({ type: 'STORIES_FETCH_INIT' });  
  
    const result = await axios.get(url);  
  
    dispatchStories({  
      type: 'STORIES_FETCH_SUCCESS',  
      payload: result.data.hits,  
    });  
  }, [url]);  
  
  ...  
};
```

To use `async/await`, our function requires the `async` keyword. Once you start using the `await` keyword on returned promises, everything reads like synchronous code. Actions after the `await` keyword are not executed until the promise resolves, meaning the code will wait. To include error handling as before, the `try` and `catch` blocks are there to help. If something goes wrong in the `try` block, the code will jump into the `catch` block to handle the error. `then/catch` blocks and `async/await` with `try/catch` blocks are both valid for handling asynchronous data in JavaScript and React.

src/App.js

```
const App = () => {  
  ...  
  
  const handleFetchStories = React.useCallback(async () => {  
    dispatchStories({ type: 'STORIES_FETCH_INIT' });  
  
    try {  
      const result = await axios.get(url);  
  
      dispatchStories({
```

```
        type: 'STORIES_FETCH_SUCCESS',
        payload: result.data.hits,
    });
} catch {
    dispatchStories({ type: 'STORIES_FETCH_FAILURE' });
}
}, [url]);

...
};
```

After all, using `async/await` with `try/catch` over `then/catch` makes it often more readable, because we avoid using callback functions and instead try to make our code more readable in a synchronous way.

Exercises:

- Confirm your [source code](#)¹⁹¹.
 - Confirm the [changes](#)¹⁹².
- Read more about [data fetching in React](#)¹⁹³.
- Read more about [async/await in JavaScript](#)¹⁹⁴.
- Optional: [Rate this section](#)¹⁹⁵.

¹⁹¹<https://bit.ly/3po4jsf>

¹⁹²<https://bit.ly/3G4t3LV>

¹⁹³<https://www.robinwieruch.de/react-hooks-fetch-data>

¹⁹⁴<https://mzl.la/3AWyWaw>

¹⁹⁵<https://forms.gle/mtMmwrrsiwioZ8GH6>

Forms in React

Earlier we introduced a new button to fetch data explicitly with a button click. We'll advance its use with a proper HTML form, which encapsulates the button and input field for the search term with its label. Forms aren't much different in React's JSX than in HTML. We'll implement it in two refactoring steps with some HTML/JavaScript. First, wrap the input field and button into an HTML form element:

src/App.js

```
const App = () => {
  ...

  return (
    <div>
      <h1>My Hacker Stories</h1>

      <form onSubmit={handleSearchSubmit}>
        <InputWithLabel
          id="search"
          value={searchTerm}
          isFocused
          onChange={handleSearchInput}
        >
          <strong>Search: </strong>
        </InputWithLabel>

        <button type="submit" disabled={!searchTerm}>
          Submit
        </button>
      </form>

      <hr />

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

Instead of passing the `handleSearchSubmit` handler to the button, it's used in the new form element. The button receives a new `type` attribute called `submit`, which indicates that the form element handles the click and not the button. Since the handler is used for the form event, it executes

`preventDefault` in React's synthetic event. This prevents the HTML form's native behavior, which leads to a browser reload:

src/App.js

```
const App = () => {
  ...

  const handleSearchSubmit = (event) => {
    setUrl(`${API_ENDPOINT}${searchTerm}`);

    event.preventDefault();
  };

  ...
};
```

Now we can execute the search feature with the keyboard's "Enter" key, because we are using a form instead of just a standalone button. In the next two steps, we will only separate the component into its own `SearchForm` component:

src/App.js

```
const SearchForm = ({
  searchTerm,
  onSearchInput,
  onSearchSubmit,
}) => (
  <form onSubmit={onSearchSubmit}>
    <InputWithLabel
      id="search"
      value={searchTerm}
      isFocused
      onChange={onSearchInput}
    >
      <strong>Search:</strong>
    </InputWithLabel>

    <button type="submit" disabled={!searchTerm}>
      Submit
    </button>
  </form>
);
```

The new component is used by the App component. However, the App component still manages the state for the form, because the state is used in the App component to fetch data passed as props (`stories.data`) to the List component:

src/App.js

```
const App = () => {
  ...

  return (
    <div>
      <h1>My Hacker Stories</h1>

      <SearchForm
        searchTerm={searchTerm}
        onSearchInput={handleSearchInput}
        onSearchSubmit={handleSearchSubmit}
      />

      <hr />

      {stories.isError && <p>Something went wrong ...</p>}

      {stories.isLoading ? (
        <p>Loading ...</p>
      ) : (
        <List list={stories.data} onRemoveItem={handleRemoveStory} />
      )}
    </div>
  );
};
```

Forms aren't much different in React than in plain HTML. When we have input fields and a button to submit data from them, we can give our HTML more structure by wrapping it into a form element with a `onSubmit` handler. The button that executes the submission needs only the "submit" type. After all, it makes it more accessible for keyboard users as well.

Exercises:

- Confirm your [source code](#)¹⁹⁶.
 - Confirm the [changes](#)¹⁹⁷.

¹⁹⁶<https://bit.ly/3jfEcjd>

¹⁹⁷<https://bit.ly/3G43M4I>

- Try what happens without using `preventDefault`.
 - Read more about [preventDefault for events in React](#)¹⁹⁸.
- Optional: [Rate this section](#)¹⁹⁹.

¹⁹⁸<https://www.robinwieruch.de/react-preventdefault>

¹⁹⁹<https://forms.gle/d14Mf7WzetP25jxq5>

React's Legacy

React has changed a lot since 2013. The iterations of its library, how React applications are written, and especially its components have all changed drastically. However, many React applications were built over the last few years, so not everything was created with the current status quo in mind. This section of the book covers React's legacy. However, I won't cover all that's considered legacy in React, because some features have been revamped more than once.

Throughout this section, we will compare a [modern React application](#)²⁰⁰ to its [legacy version](#)²⁰¹. We'll discover that most differences between modern and legacy React are due to class components versus function components.

²⁰⁰<https://bit.ly/3vrKgu9>

²⁰¹<https://bit.ly/2Z1matY>

React Class Components

React components have undergone many changes, from **createClass components** over **class components**, to **function components**. Going through a React application today, it's likely that we'll see class components next to the modern function components. Fortunately, you will not see many **createClass components** anymore.

A typical class component is a JavaScript class with a mandatory **render method** that returns the JSX. The class extends from a `React.Component` to inherit ([class inheritance](#)²⁰²) all React's component features (e.g. state management, lifecycle methods for side-effects). React props are accessed via the class instance (`this`) in any class method (e.g. `render`):

Code Playground

```
class InputWithLabel extends React.Component {
  render() {
    const {
      id,
      value,
      type = 'text',
      onChange,
      children,
    } = this.props;

    return (
      <>
        <label htmlFor={id}>{children}</label>
        &nbsp;
        <input
          id={id}
          type={type}
          value={value}
          onChange={onChange}
        />
      </>
    );
  }
}
```

For a while, class components were the popular choice for writing React applications. Eventually, function components were added, and both co-existed with their distinct purposes side by side:

²⁰²<https://bit.ly/3vxh3xY>

Code Playground

```
const InputWithLabel = ({
  id,
  value,
  type = 'text',
  onChange,
  children,
}) => (
  <>
    <label htmlFor={id}>{children}</label>
    &nbsp;
    <input
      id={id}
      type={type}
      value={value}
      onChange={onChange}
    />
  </>
);
```

If no side-effects and no state were used in legacy apps, we'd use a function component instead of a class component. Before 2018 – before React Hooks were introduced – React's function components couldn't handle side-effects (`useEffect` hooks) or state (`useState/useReducer` hooks). As a result, these components were known as **functional stateless components**, there only to input props and output JSX. To use state or side-effects, it was necessary to refactor from a function component to a class component. When neither state nor side-effects were used, we used class components or more commonly the more lightweight function component.

With the addition of React Hooks, function components were granted the same features as class components, with state and side-effects. And since there was no longer any practical difference between them, the community chose function components over class components since they are more lightweight.

Exercises:

- Read more about [JavaScript Classes](#)²⁰³.
- Read more about [how to refactor from a class component to a function component](#)²⁰⁴.
- Learn more about a different [class component syntax](#)²⁰⁵ which wasn't popular but more effective.

²⁰³<https://mzl.la/3vvc2FO>

²⁰⁴<https://www.robinwieruch.de/react-hooks-migration>

²⁰⁵<https://bit.ly/3lYzrfT>

- Read more about [class components in depth](#)²⁰⁶.
- Read more about [other legacy and modern component types in React](#)²⁰⁷.
- Optional: [Rate this section](#)²⁰⁸.

²⁰⁶<https://bit.ly/3FXUibf>

²⁰⁷<https://www.robinwieruch.de/react-component-types>

²⁰⁸<https://forms.gle/g5qLH1KZ5Y1JE1v57>

React Class Components: State

Before React Hooks, class components were superior to function components because they could be stateful. With a class constructor, we can set an initial state for the component. Also, the component's instance (`this`) gives access to the current state (`this.state`) and the component's state updater method (`this.setState`):

Code Playground

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

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

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

    return (
      <div>
        <h1>My Hacker Stories</h1>

        <SearchForm
          searchTerm={searchTerm}
          onSearchInput={() => this.setState({
            searchTerm: event.target.value
          })}
        />
      </div>
    );
  }
}
```

If the state has more than one property in its state object, the `setState` method performs only a shallow update. Only the properties passed to `setState` are overwritten, and all other properties in the state object stay intact. Since state management is important for frontend applications, there was no way around class components without hooks for function components.

In a React class component, there are two dedicated APIs (`this.state` and `this.setState`) to manage a component's state. In a function component, React's `useState` and `useReducer` hooks

handle this. Related items are packed into one state hook, while a class component must use a general state API. This was one of the major reasons to introduce React Hooks and move away from class components.

Exercises:

- Write a stateful class component (e.g. Input component where a user can type something into it and the text displays somewhere else).
- Optional: [Rate this section](https://forms.gle/5UPEsMWSGjWyeaiA9)²⁰⁹.

²⁰⁹<https://forms.gle/5UPEsMWSGjWyeaiA9>

Imperative React

In a React function component, React's `useRef` Hook is used mostly for imperative programming. Throughout React's history, the *ref* and its usage had a few versions:

- String Refs (deprecated)
- Callback Refs (used in class and function components)
- `createRef` Refs (exclusive for class components)
- `useRef` Hook Refs (exclusive for function components)

The React team introduced **React's `createRef`** with version 16.3 as the latest equivalent to a function component's `useRef` hook which has been integrated with React Hooks in 16.8:

Code Playground

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

    this.inputRef = React.createRef();
  }

  componentDidMount() {
    if (this.props.isFocused) {
      this.inputRef.current.focus();
    }
  }

  render() {
    ...

    return (
      <>
        ...
        <input
          ref={this.inputRef}
          id={id}
          type={type}
          value={value}
          onChange={onInputChange}
        />
      </>
    );
  }
}
```

```
}  
}
```

With the helper function, the ref is created in the class' constructor, applied in the JSX for the ref attribute, and here used in a class component's lifecycle method (equivalent to a function component's useEffect Hook). The ref can also be used elsewhere, like focusing the input field on a button click.

Where createRef is used in React's class components, React's useRef Hook is used in React function components. As React shifts towards function components, today it's common practice to use the useRef hook exclusively to manage refs and apply imperative programming principles.

Exercises:

- Read more about [the different ref techniques in React](#)²¹⁰.
- Optional: [Rate this section](#)²¹¹.

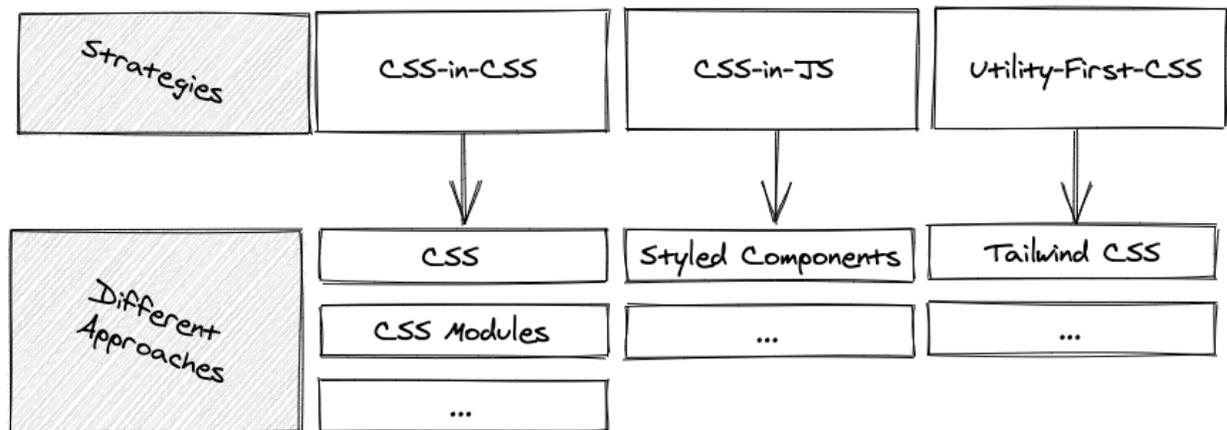
²¹⁰<https://bit.ly/3jjhCXa>

²¹¹<https://forms.gle/xd131YRH6NdZa7p7>

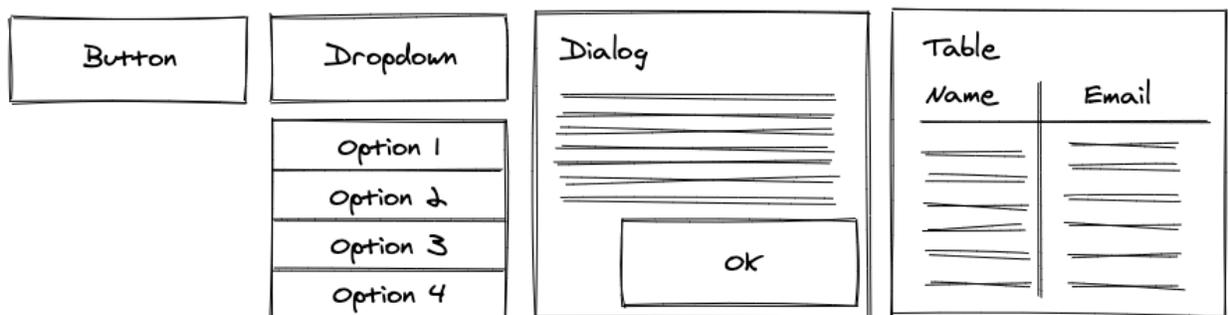
Styling in React

There are many ways to style a React application, and there are lengthy debates about the best **styling strategy** and **styling approach**. We'll go over a few of these approaches without giving them too much weight. There will be some pro and con arguments, but it's mostly a matter of what fits best for developers and teams.

We will begin React styling with common CSS in React, but then explore two alternatives for more advanced **CSS-in-CSS (CSS Modules)** and **CSS-in-JS (Styled Components)** strategies. CSS Modules and Styled Components are only two approaches out of many in both groups of strategies. We'll also cover how to include scalable vector graphics (SVGs), such as a logo or icons, in our React application.



If you don't want to build common UI components (e.g. button, dialog, dropdown) from scratch, you can always pick a [popular UI library suited for React](https://www.robinwieruch.de/react-libraries)²¹², which provides these components by default. However, it is better for learning React if you try building these components before using a pre-built solution. As a result, we won't use any of the UI component libraries.



²¹²<https://www.robinwieruch.de/react-libraries>

The following styling approaches and SVGs are pre-configured in `create-react-app`. If you're in control of the build tools (e.g. Webpack) by having a custom setup, they might need to be configured to enable importing CSS or SVG files. Since we are using `create-react-app`, we can use these files as assets right away.

Exercises:

- Read more about [the different styling strategies and approaches in React](https://www.robinwieruch.de/react-css-styling)²¹³.

²¹³<https://www.robinwieruch.de/react-css-styling>

CSS in React

Common CSS in React is similar to the standard CSS you may have already learned. Each web application gives HTML elements a `class` (in React it's `className`) attribute that is styled via a CSS file:

`src/App.js`

```
const App = () => {
  ...

  return (
    <div className="container">
      <h1 className="headline-primary">My Hacker Stories</h1>

      <SearchForm
        searchTerm={searchTerm}
        onSearchInput={handleSearchInput}
        onSearchSubmit={handleSearchSubmit}
      />

      {stories.isError && <p>Something went wrong ...</p>}

      {stories.isLoading ? (
        <p>Loading ...</p>
      ) : (
        <List list={stories.data} onRemoveItem={handleRemoveStory} />
      )}
    </div>
  );
};
```

The `<hr />` was removed because the CSS handles the border in the next steps. We'll import the CSS file, which is done with the help of the create-react-app configuration:

`src/App.js`

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

import './App.css';
```

This CSS file will define the two (and more) CSS classes we used (and will use) in the App component. In your `src/App.css` file, define them like the following:

src/App.css

```
.container {
  height: 100vw;
  padding: 20px;

  background: #83a4d4; /* fallback for old browsers */
  background: linear-gradient(to left, #b6fbff, #83a4d4);

  color: #171212;
}

.headline-primary {
  font-size: 48px;
  font-weight: 300;
  letter-spacing: 2px;
}
```

You should see the first stylings taking effect in your application when you start it again. Next, we will head over to the Item component. Some of its elements receive the `className` attribute too, however, we are also using a new styling technique here:

src/App.js

```
const Item = ({ item, onRemoveItem }) => (
  <li className="item">
    <span style={{ width: '40%' }}>
      <a href={item.url}>{item.title}</a>
    </span>
    <span style={{ width: '30%' }}>{item.author}</span>
    <span style={{ width: '10%' }}>{item.num_comments}</span>
    <span style={{ width: '10%' }}>{item.points}</span>
    <span style={{ width: '10%' }}>
      <button
        type="button"
        onClick={() => onRemoveItem(item)}
        className="button button_small"
      >
        Dismiss
      </button>
    </span>
  </li>
);
```

As you can see, we can also use the `style` attribute for HTML elements. In JSX, style can be passed as an inline JavaScript object to these attributes. This way we can define dynamic style properties in JavaScript files rather than mostly static CSS files. This approach is called **inline style**, which is useful for quick prototyping and dynamic style definitions. Inline style should be used sparingly, however, since a separate style definition with a CSS file keeps the JSX more concise.

In your `src/App.css` file, define the new CSS classes. Basic CSS features are used here, because advanced CSS features (e.g. nesting) from CSS extensions (e.g. Sass) are not included in this example, as they are [optional configurations](#)²¹⁴:

`src/App.css`

```
.item {
  display: flex;
  align-items: center;
  padding-bottom: 5px;
}

.item > span {
  padding: 0 5px;
  white-space: nowrap;
  overflow: hidden;
  white-space: nowrap;
  text-overflow: ellipsis;
}

.item > span > a {
  color: inherit;
}
```

The button style from the previous component is still missing, so we'll define a base button style and two more specific button styles (small and large). One of the button specifications has been already used, the other will be used in the next steps:

²¹⁴<https://bit.ly/3E1a2bM>

src/App.css

```
.button {
  background: transparent;
  border: 1px solid #171212;
  padding: 5px;
  cursor: pointer;

  transition: all 0.1s ease-in;
}

.button:hover {
  background: #171212;
  color: #ffffff;
}

.button_small {
  padding: 5px;
}

.button_large {
  padding: 10px;
}
```

Apart from styling approaches in React, naming conventions ([CSS guidelines](#)²¹⁵) are a whole other topic. The last CSS snippet followed BEM rules by defining modifications of a class with an underscore (_). Choose whatever naming convention suits you and your team. Without further ado, we will style the next React component:

src/App.js

```
const SearchForm = ({ ... }) => (
  <form onSubmit={onSearchSubmit} className="search-form">
    <InputWithLabel ... >
      <strong>Search:</strong>
    </InputWithLabel>

    <button
      type="submit"
      disabled={!searchTerm}
      className="button button_large"
    >
      Submit
```

²¹⁵<https://mzl.la/3m5avnb>

```
    </button>
  </form>
);
```

We can also pass the `className` attribute as a prop to React components. For example, we can use this option to pass the `SearchForm` component a flexible style with a `className` prop from a range of predefined classes (e.g. `button_large` or `button_small`) from a CSS file. Lastly, style the `InputWithLabel` component:

src/App.js

```
const InputWithLabel = ({ ... }) => {
  ...

  return (
    <>
      <label htmlFor={id} className="label">
        {children}
      </label>
      &nbsp;
      <input
        ref={inputRef}
        id={id}
        type={type}
        value={value}
        onChange={onInputChange}
        className="input"
      />
    </>
  );
};
```

In your `src/App.css` file, add the remaining classes:

src/App.css

```
.search-form {
  padding: 10px 0 20px 0;
  display: flex;
  align-items: baseline;
}

.label {
  border-top: 1px solid #171212;
  border-left: 1px solid #171212;
  padding-left: 5px;
  font-size: 24px;
}

.input {
  border: none;
  border-bottom: 1px solid #171212;
  background-color: transparent;

  font-size: 24px;
}
```

For simplicity, we styled elements like label and input individually in the `src/App.css` file. However, in a real application, it may be better to define these elements once in the `src/index.css` file globally. As React components are split into multiple files, sharing style becomes a necessity. After all, this is the basic usage of CSS in React. Without CSS extensions like Sass (Syntactically Awesome Style Sheets), styling can become more burdensome, though, because features like CSS nesting are not available in native CSS.

Exercises:

- Confirm your [source code](#)²¹⁶.
 - Confirm the [changes](#)²¹⁷.
- Read more about [CSS stylesheets in create-react-app](#)²¹⁸.
 - Try to pass `className` prop from App to SearchForm component, either with the value `button_small` or `button_large`, and use this as `className` for the button element.
- Read more about [Sass in create-react-app](#)²¹⁹ for taking advantage of more advanced CSS features like nesting.
- Enable Sass in your create-react-app and start to use its features (e.g. nesting) in your CSS file.
- Optional: [Rate this section](#)²²⁰.

²¹⁶<https://bit.ly/3jk9llQ>

²¹⁷<https://bit.ly/3vrrnr4>

²¹⁸<https://bit.ly/3G0fTjn>

²¹⁹<https://bit.ly/3GeJGF9>

²²⁰<https://forms.gle/RovYbjYF9McD1h6c7>

CSS Modules in React

CSS Modules are an advanced CSS-in-CSS approach. The CSS file stays the same, where you could apply CSS extensions like Sass, but its use in React components changes. To enable CSS modules in create-react-app, rename the `src/App.css` file to `src/App.module.css`. This action is performed in the command line from your project's directory:

Command Line

```
mv src/App.css src/App.module.css
```

In the renamed `src/App.module.css`, start with the first CSS class definitions, as before:

src/App.module.css

```
.container {
  height: 100vw;
  padding: 20px;

  background: #83a4d4; /* fallback for old browsers */
  background: linear-gradient(to left, #b6fbff, #83a4d4);

  color: #171212;
}

.headlinePrimary {
  font-size: 48px;
  font-weight: 300;
  letter-spacing: 2px;
}
```

Import the `src/App.module.css` file with a relative path again. This time, import it as a JavaScript object where the name of the object (here `styles`) is up to you:

src/App.js

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

import styles from './App.module.css';
```

Instead of defining the `className` as a string mapped to a CSS file, access the CSS class directly from the `styles` object, and assign it with a JavaScript in JSX expression to your elements.

src/App.js

```

const App = () => {
  ...

  return (
    <div className={styles.container}>
      <h1 className={styles.headlinePrimary}>My Hacker Stories</h1>

      <SearchForm
        searchTerm={searchTerm}
        onSearchInput={handleSearchInput}
        onSearchSubmit={handleSearchSubmit}
      />

      {stories.isError && <p>Something went wrong ...</p>}

      {stories.isLoading ? (
        <p>Loading ...</p>
      ) : (
        <List list={stories.data} onRemoveItem={handleRemoveStory} />
      )}
    </div>
  );
};

```

There are various ways to add multiple CSS classes via the `styles` object to the element's single `className` attribute. Here, we use JavaScript template literals:

src/App.js

```

const Item = ({ item, onRemoveItem }) => (
  <li className={styles.item}>
    <span style={{ width: '40%' }}>
      <a href={item.url}>{item.title}</a>
    </span>
    <span style={{ width: '30%' }}>{item.author}</span>
    <span style={{ width: '10%' }}>{item.num_comments}</span>
    <span style={{ width: '10%' }}>{item.points}</span>
    <span style={{ width: '10%' }}>
      <button
        type="button"
        onClick={() => onRemoveItem(item)}
        className={` ${styles.button} ${styles.buttonSmall} `}

```

```
    >  
      Dismiss  
    </button>  
  </span>  
</li>  
);
```

We can also add inline styles as more dynamic styles in JSX again. It's also possible to add a CSS extension like Sass to enable advanced features like CSS nesting (see the previous section). We will stick to native CSS features though:

src/App.module.css

```
.item {  
  display: flex;  
  align-items: center;  
  padding-bottom: 5px;  
}  
  
.item > span {  
  padding: 0 5px;  
  white-space: nowrap;  
  overflow: hidden;  
  white-space: nowrap;  
  text-overflow: ellipsis;  
}  
  
.item > span > a {  
  color: inherit;  
}
```

Then the button CSS classes in the *src/App.module.css* file:

src/App.module.css

```
.button {  
  background: transparent;  
  border: 1px solid #171212;  
  padding: 5px;  
  cursor: pointer;  
  
  transition: all 0.1s ease-in;  
}
```

```
.button:hover {  
  background: #171212;  
  color: #ffffff;  
}  
  
.buttonSmall {  
  padding: 5px;  
}  
  
.buttonLarge {  
  padding: 10px;  
}
```

There is a shift toward pseudo BEM naming conventions here, in contrast to `button_small` and `button_large` from the previous section. If the previous naming convention holds true, we can only access the style with `styles['button_small']` which makes it more verbose because of JavaScript's limitation with object underscores. The same shortcomings would apply for classes defined with a dash (-). In contrast, now we can use `styles.buttonSmall` instead (see: Item component):

`src/App.js`

```
const SearchForm = ({ ... }) => (  
  <form onSubmit={onSearchSubmit} className={styles.searchForm}>  
    <InputWithLabel ... >  
      <strong>Search:</strong>  
    </InputWithLabel>  
  
    <button  
      type="submit"  
      disabled={!searchTerm}  
      className={` ${styles.button} ${styles.buttonLarge} `}  
    >  
      Submit  
    </button>  
  </form>  
)
```

The `SearchForm` component receives the styles as well. It has to use string interpolation for using two styles in one element via JavaScript's template literals. One alternative way is the `classnames`²²¹ library, which is installed via the command line as a project dependency:

²²¹<https://bit.ly/3aSz1Bl>

src/App.js

```
import cs from 'classnames';

...

// somewhere in a className attribute
className={cs(styles.button, styles.buttonLarge)}
```

The library offers conditional styling too; whereas the left-hand side of the object's property must be used as a [computed property name](#)²²² and is only applied if the right-hand side evaluates to true:

src/App.js

```
import cs from 'classnames';

...

// somewhere in a className attribute
className={cs(styles.button, { [styles.buttonLarge]: isLarge })}
```

Finally, continue with the InputWithLabel component:

src/App.js

```
const InputWithLabel = ({ ... }) => {
  ...

  return (
    <>
      <label htmlFor={id} className={styles.label}>
        {children}
      </label>
      &nbsp;
      <input
        ref={inputRef}
        id={id}
        type={type}
        value={value}
        onChange={onInputChange}
        className={styles.input}
      />
    </>
  )
}
```

²²²<https://mzl.la/2XuN651>

```
);  
};
```

And finish up the remaining style in the `src/App.module.css` file:

`src/App.module.css`

```
.searchForm {  
  padding: 10px 0 20px 0;  
  display: flex;  
  align-items: baseline;  
}  
  
.label {  
  border-top: 1px solid #171212;  
  border-left: 1px solid #171212;  
  padding-left: 5px;  
  font-size: 24px;  
}  
  
.input {  
  border: none;  
  border-bottom: 1px solid #171212;  
  background-color: transparent;  
  
  font-size: 24px;  
}
```

The same caution as the last section applies: some of these styles like `input` and `label` might be more efficient in a global `src/index.css` file without CSS modules.

Again, CSS Modules – like any other CSS-in-CSS approach – can use Sass for more advanced CSS features like nesting. The advantage of CSS modules is that we receive an error in JavaScript each time a style isn't defined. In the standard CSS approach, unmatched styles in JavaScript and CSS files might go unnoticed.

Exercises:

- Confirm your [source code](#)²²³.
 - Confirm the [changes](#)²²⁴.
- Read more about [CSS Modules in create-react-app](#)²²⁵.
- Optional: [Rate this section](#)²²⁶.

²²³<https://bit.ly/3lVeob>

²²⁴<https://bit.ly/3pkqgs4>

²²⁵<https://bit.ly/3phgN51>

²²⁶<https://forms.gle/iuU7WaeJVwHN2pFCA>

Styled Components in React

With the previous approaches from CSS-in-CSS, Styled Components is one of several approaches for CSS-in-JS. I picked Styled Components because it's the most popular. It comes as a JavaScript dependency, so we must install it on the command line:

Command Line

```
npm install styled-components
```

Then import it in your *src/App.js* file:

src/App.js

```
import * as React from 'react';  
import axios from 'axios';  
import styled from 'styled-components';
```

As the name suggests, CSS-in-JS happens in your JavaScript file. In your *src/App.js* file, define your first styled components:

src/App.js

```
const StyledContainer = styled.div`  
  height: 100vw;  
  padding: 20px;  
  
  background: #83a4d4;  
  background: linear-gradient(to left, #b6fbff, #83a4d4);  
  
  color: #171212;  
`;  
  
const StyledHeadlinePrimary = styled.h1`  
  font-size: 48px;  
  font-weight: 300;  
  letter-spacing: 2px;  
`;
```

When using Styled Components, you are using the JavaScript template literals the same way as JavaScript functions. Everything between the backticks can be seen as an argument and the `styled` object gives you access to all the necessary HTML elements (e.g. `div`, `h1`) as functions. Once a function is called with the style, it returns a React component that can be used in your App component:

src/App.js

```
const App = () => {
  ...

  return (
    <StyledContainer>
      <StyledHeadlinePrimary>My Hacker Stories</StyledHeadlinePrimary>

      <SearchForm
        searchTerm={searchTerm}
        onSearchInput={handleSearchInput}
        onSearchSubmit={handleSearchSubmit}
      />

      {stories.isError && <p>Something went wrong ...</p>}

      {stories.isLoading ? (
        <p>Loading ...</p>
      ) : (
        <List list={stories.data} onRemoveItem={handleRemoveStory} />
      )}
    </StyledContainer>
  );
};
```

This kind of React component follows the same rules as a common React component. Everything passed between its element tags is passed automatically as React children prop. For the Item component, we are not using inline styles this time, but defining a dedicated styled component for it. `StyledColumn` receives its styles dynamically using a React prop:

src/App.js

```
const Item = ({ item, onRemoveItem }) => (
  <StyledItem>
    <StyledColumn width="40%">
      <a href={item.url}>{item.title}</a>
    </StyledColumn>
    <StyledColumn width="30%">{item.author}</StyledColumn>
    <StyledColumn width="10%">{item.num_comments}</StyledColumn>
    <StyledColumn width="10%">{item.points}</StyledColumn>
    <StyledColumn width="10%">
      <StyledButtonSmall
        type="button"
```

```
      onClick={() => onRemoveItem(item)}
    >
      Dismiss
    </StyledButtonSmall>
  </StyledColumn>
</StyledItem>
);
```

The flexible `width` prop is accessible in the styled component's template literal as an argument of an inline function. The return value from the function is applied there as a string. Since we can use immediate returns when omitting the arrow function's body, it becomes a concise inline function:

src/App.js

```
const StyledItem = styled.li`
  display: flex;
  align-items: center;
  padding-bottom: 5px;
`;

const StyledColumn = styled.span`
  padding: 0 5px;
  white-space: nowrap;
  overflow: hidden;
  white-space: nowrap;
  text-overflow: ellipsis;

  a {
    color: inherit;
  }

  width: ${(props) => props.width};
`;
```

Advanced features like CSS nesting are available in Styled Components by default. Nested elements are accessible and the current element can be selected with the `&` CSS operator:

src/App.js

```
const StyledButton = styled.button`
  background: transparent;
  border: 1px solid #171212;
  padding: 5px;
  cursor: pointer;

  transition: all 0.1s ease-in;

  &:hover {
    background: #171212;
    color: #ffffff;
  }
`;
```

You can also create specialized versions of styled components by passing another component to the library's function. The specialized button receives all the base styles from the previously defined `StyledButton` component:

src/App.js

```
const StyledButtonSmall = styled(StyledButton)`
  padding: 5px;
`;

const StyledButtonLarge = styled(StyledButton)`
  padding: 10px;
`;

const StyledSearchForm = styled.form`
  padding: 10px 0 20px 0;
  display: flex;
  align-items: baseline;
`;
```

When we use a styled component like `StyledSearchForm`, its underlying form element is used in the real HTML output. We can continue using the native HTML attributes (`onSubmit`, `type`, `disabled`) there:

src/App.js

```
const SearchForm = ({ ... }) => (  
  <StyledSearchForm onSubmit={onSearchSubmit}>  
    <InputWithLabel  
      id="search"  
      value={searchTerm}  
      isFocused  
      onChange={onSearchInput}  
    >  
      <strong>Search:</strong>  
    </InputWithLabel>  
  
    <StyledButtonLarge type="submit" disabled={!searchTerm}>  
      Submit  
    </StyledButtonLarge>  
  </StyledSearchForm>  
)  
);
```

Finally, the InputWithLabel decorated with its yet undefined styled components:

src/App.js

```
const InputWithLabel = ({ ... }) => {  
  ...  
  
  return (  
    <>  
      <StyledLabel htmlFor={id}>{children}</StyledLabel>  
      &nbsp;  
      <StyledInput  
        ref={inputRef}  
        id={id}  
        type={type}  
        value={value}  
        onChange={onInputChange}  
      />  
    </>  
  );  
};
```

And its matching styled components are defined in the same file:

src/App.js

```
const StyledLabel = styled.label`
  border-top: 1px solid #171212;
  border-left: 1px solid #171212;
  padding-left: 5px;
  font-size: 24px;
`;

const StyledInput = styled.input`
  border: none;
  border-bottom: 1px solid #171212;
  background-color: transparent;

  font-size: 24px;
`;
```

CSS-in-JS with styled components shifts the focus of defining styles to actual React components. Styled Components are styles defined as React components without the intermediate CSS file. If a styled component isn't used in a JavaScript, your IDE/editor will tell you. Styled Components are bundled next to other JavaScript assets in JavaScript files for a production-ready application. There are no extra CSS files, but only JavaScript when using the CSS-in-JS strategy. Both strategies, CSS-in-JS and CSS-in-CSS, and their approaches (e.g. Styled Components and CSS Modules) are popular among React developers. Use what suits you and your team best.

Exercises:

- Confirm your [source code](#)²²⁷.
 - Confirm the [changes](#)²²⁸.
- Read more about [best practices for Styled Components in React](#)²²⁹.
- Usually there is no *src/index.css* file for global styles when using Styled Components. Find out how to use global styles when using Styled Components.
- Optional: [Rate this section](#)²³⁰.

²²⁷<https://bit.ly/2ZaHeOU>

²²⁸<https://bit.ly/3n4aH5e>

²²⁹<https://www.robinwieruch.de/styled-components>

²³⁰<https://forms.gle/5vFxxvg9hSNAna37S8>

SVGs in React

To create a modern React application, we'll likely need to use SVGs. Instead of giving every button element text, for example, we might want to make it lightweight with an icon. In this section, we'll use a scalable vector graphic (SVG) as an icon in one of our React components.

This section builds on the "CSS in React" we covered earlier, to give the SVG icon a good look and feel right away. It's acceptable to use a different styling approach, or no styling at all, though the SVG might look off without it.

This icon as SVG is taken from [Flaticon's Freepick](#)²³¹. Many of the SVGs on this website are free to use, though they require you to mention the author. You can download the icon from [here](#)²³² as SVG and put it in your project as `src/check.svg`. Downloading the file is the recommended way, however, for the sake of completion, this is the verbose SVG definition:

Code Playground

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!-- Generator: Adobe Illustrator 18.0.0, SVG Export Plug-In . SVG Version: 6.00 Bui\
ld 0) -->
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN" "http://www.w3.org/Graphics/SVG/1.1/D\
TD/svg11.dtd">
<svg version="1.1" id="Capa_1" xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http:\
//www.w3.org/1999/xlink" x="0px" y="0px"
  viewBox="0 0 297 297" style="enable-background:new 0 0 297 297;" xml:space="prese\
rve">
  <g>
    <path d="M113.636,272.638c-2.689,0-5.267-1.067-7.168-2.97L2.967,166.123c-3.956-3\
.957-3.956-10.371-0.001-14.329154.673-54.703
      c1.9-1.9,4.479-2.97,7.167-2.97c2.689,0,5.268,1.068,7.169,2.969141.661,41.676L2\
25.023,27.332c1.9-1.901,4.48-2.97,7.168-2.9710,0
      c2.688,0,5.268,1.068,7.167,2.97154.675,54.701c3.956,3.957,3.956,10.372,0,14.32\
8L120.803,269.668
      C118.903,271.57,116.325,272.638,113.636,272.638z M24.463,158.958189.173,89.209\
1158.9-158.971-40.346-40.364L120.803,160.264
      c-1.9,1.902-4.478,2.971-7.167,2.971c-2.688,0-5.267-1.068-7.168-2.971-41.66-41.\
674L24.463,158.958z"/>
  </g>
</svg>
```

Because we're using `create-react-app` again, we can import SVGs (similar to CSS) as React components right away. In `src/App.js`, use the following syntax for importing the SVG:

²³¹<https://bit.ly/3E16SEz>

²³²<https://bit.ly/2Z2EoeA>

src/App.js

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

import './App.css';
import { ReactComponent as Check } from './check.svg';
```

We are importing an SVG, and this works for many different uses for SVGs (e.g. logo, background). Instead of a button text, pass the SVG component with a `height` and `width` attribute:

src/App.js

```
const Item = ({ item, onRemoveItem }) => (
  <li className="item">
    <span style={{ width: '40%' }}>
      <a href={item.url}>{item.title}</a>
    </span>
    <span style={{ width: '30%' }}>{item.author}</span>
    <span style={{ width: '10%' }}>{item.num_comments}</span>
    <span style={{ width: '10%' }}>{item.points}</span>
    <span style={{ width: '10%' }}>
      <button
        type="button"
        onClick={() => onRemoveItem(item)}
        className="button button_small"
      >
        <Check height="18px" width="18px" />
      </button>
    </span>
  </li>
);
```

Regardless of the styling approach you are using, you can give your SVG icon in the button a hover effect too. In the basic CSS approach, it would look like the following in the `src/App.css` file:

src/App.css

```
.button:hover > svg > g {
  fill: #ffffff;
  stroke: #ffffff;
}
```

The create-react-app project makes using SVGs straightforward, with no extra configuration needed. This is different if you create a React project from scratch with build tools like Webpack, because you have to take care of it yourself. Anyway, SVGs make your application more approachable, so use them whenever it suits you.

Exercises:

- Confirm your [source code](#)²³³.
 - Confirm the [changes](#)²³⁴.
- Read more about [SVGs in create-react-app](#)²³⁵.
- Read more about [SVG background patterns in React](#)²³⁶.
- Add another SVG icon in your application.
- Integrate the third-party library [react-fontawesome](#)²³⁷ or [react-icons](#)²³⁸ into your application and use its SVG symbols.
- Optional: [Rate this section](#)²³⁹.

²³³<https://bit.ly/3DSltSI>

²³⁴<https://bit.ly/3lYaEZf>

²³⁵<https://bit.ly/3DVxBT9>

²³⁶<https://www.robinwieruch.de/react-svg-patterns>

²³⁷<https://bit.ly/2ZdNXI7>

²³⁸<https://bit.ly/3nayoJ7>

²³⁹<https://forms.gle/3yGgMDR2VQ5WksSXA>

React Maintenance

Once a React application grows, maintenance becomes a priority. To prepare for this eventuality, we'll cover performance optimization, type safety, testing, and project structure. Each of these topics will strengthen your app to take on more functionality without losing quality.

Performance optimization prevents applications from slowing down by assuring efficient use of available resource. Typed programming languages like TypeScript detect bugs earlier in the feedback loop. Testing gives us more explicit feedback than typed programming, and provides a way to understand which actions can break the application. Lastly, a project structure supports the organized management of assets into folders and files, which is especially useful in scenarios where team members work in different domains.

Performance in React (Advanced)

This section is just here for the sake of learning about performance improvements in React. We wouldn't need optimizations in most React applications, as React is fast out of the box. While more sophisticated tools exist for performance measurements in JavaScript and React, we will stick to a simple `console.log()` and our browser's developer tools for the logging output.

Don't run on first render

Previously, we have covered React's `useEffect` Hook, which is used for side-effects. It runs the first time a component renders (mounting), and then every re-render (updating). By passing an empty dependency array to it as a second argument, we can tell the hook to run on the first render only. Out of the box, there is no way to tell the hook to run only on every re-render (update) and not on the first render (mount). For example, examine our custom hook for state management with React's `useState` Hook and its semi-persistent state with local storage using React's `useEffect` Hook:

`src/App.js`

```
const useSemiPersistentState = (key, initialState) => {
  const [value, setValue] = React.useState(
    localStorage.getItem(key) || initialState
  );

  React.useEffect(() => {
    console.log('A');
    localStorage.setItem(key, value);
  }, [value, key]);

  return [value, setValue];
};
```

With a closer look at the developer's tools, we can see the log for the first time when the component renders using this custom hook. It doesn't make sense to run the side-effect for the initial rendering of the component though, because there is nothing to store in the local storage except the initial value. It's a redundant function invocation, and should only run for every update (re-rendering) of the component.

As mentioned, there is no React Hook that runs on every re-render, and there is no way to tell the `useEffect` hook in a React idiomatic way to call its function only on every re-render. However, by using React's `useRef` Hook which keeps its `ref.current` property intact over re-renders, we can keep a *made up state* (without re-rendering the component on state updates) with an instance variable of our component's lifecycle:

src/App.js

```
const useSemiPersistentState = (key, initialState) => {
  const isMounted = React.useRef(false);

  const [value, setValue] = React.useState(
    localStorage.getItem(key) || initialState
  );

  React.useEffect(() => {
    if (!isMounted.current) {
      isMounted.current = true;
    } else {
      console.log('A');
      localStorage.setItem(key, value);
    }
  }, [value, key]);

  return [value, setValue];
};
```

We are exploiting the `ref` and its mutable `current` property for imperative state management that doesn't trigger a re-render. Once the hook is called from its component for the first time (component render), the `ref`'s `current` property is initialized with a `false` boolean called `isMounted`. As a result, the side-effect function in `useEffect` isn't called; only the boolean flag for `isMounted` is toggled to `true` in the side-effect. Whenever the hook runs again (component re-render), the boolean flag is evaluated in the side-effect. Since it's `true`, the side-effect function runs. Over the lifetime of the component, the `isMounted` boolean will remain `true`. It was there to avoid calling the side-effect function for the first time render that uses our custom hook.

The above was only about preventing the invocation of one simple function for a component rendering for the first time. But imagine you have an expensive computation in your side-effect, or the custom hook is used frequently in the application. It's more practical to deploy this technique to avoid unnecessary function invocations.

Exercises:

- Read more about [running useEffect only on update](#)²⁴⁰.
- Read more about [running useEffect only once](#)²⁴¹.

²⁴⁰<https://www.robinwieruch.de/react-useeffect-only-on-update>

²⁴¹<https://www.robinwieruch.de/react-useeffect-only-once>

Don't re-render if not needed

Earlier, we have explored React's re-rendering mechanism. We'll repeat this exercise for the App and List components. For both components, add a logging statement:

src/App.js

```
const App = () => {  
  ...  
  
  console.log('B:App');  
  
  return ( ... );  
};  
  
const List = ({ list, onRemoveItem }) =>  
  console.log('B:List') || (  
    <ul>  
      {list.map((item) => (  
        <Item  
          key={item.objectID}  
          item={item}  
          onRemoveItem={onRemoveItem}  
        />  
      ))}  
    </ul>  
  );
```

Because the List component has no function body, and developers are lazy folks who don't want to refactor the component for a simple logging statement, the List component uses the `||` operator instead. This is a neat trick for adding a logging statement to a function component without a function body. Since the `console.log()` on the left-hand side of the operator always evaluates to false, the right-hand side of the operator gets always executed.

Code Playground

```
function getTheTruth() {  
  if (console.log('B:List')) {  
    return true;  
  } else {  
    return false;  
  }  
}
```

```
console.log(getTheTruth());  
// B:List  
// false
```

Let's focus on the actual logging in the browser's developer tools. You should see a similar output. First, the App component renders, followed by its child components (e.g. List component).

Visualization

```
B:App  
B:List  
B:App  
B:App  
B:List
```

*Note: If you are seeing more than these loggings, check whether your `*src/index.js` file uses `<React.StrictMode>` as a wrapper for your App component. If it's the case, remove the strict mode and check your logging again. Explanation: In development mode, React's StrictMode renders a component twice to detect problems with your implementation in order to warn you about these. This StrictMode is automatically excluded for applications in production. However, if you don't want to be confused by the multiple renders, remove StrictMode from the `src/index.js` file.**

Since a side-effect triggers data fetching after the first render, only the App component renders, because the List component is replaced by a loading indicator in a conditional rendering. Once the data arrives, both components render again.

Visualization

```
// initial render  
B:App  
B:List  
  
// data fetching with loading  
B:App  
  
// re-rendering with data  
B:App  
B:List
```

So far, this behavior is acceptable, since everything renders on time. Now we'll take this experiment a step further, by typing into the SearchForm component's input field. You should see the changes with every character entered into the element:

Visualization

B:App

B:List

What's striking is that the List component shouldn't re-render, but it does. The search feature isn't executed via its button, so the `list` passed to the List component remains the same for every keystroke. This is React's default behavior, re-rendering everything below a component with a state change, which surprises many people. In other words, if a parent component re-renders, its child components re-render as well. React does this by default, because preventing a re-render of child components could lead to bugs, and the re-rendering mechanism of React is often fast enough by default..

Sometimes we want to prevent re-rendering, however. For example, huge data sets displayed in a table shouldn't re-render if they are not affected by an update. It's more efficient to perform an equality check if something changed for the component. Therefore, we can use React's memo API to make this equality check for the props:

src/App.js

```
const List = React.memo(
  ({ list, onRemoveItem }) =>
    console.log('B:List') || (
      <ul>
        {list.map((item) => (
          <Item
            key={item.objectID}
            item={item}
            onRemoveItem={onRemoveItem}
          />
        ))}
      </ul>
    )
);
```

However, the output stays the same when typing into the SearchForm's input field:

Visualization

B:App

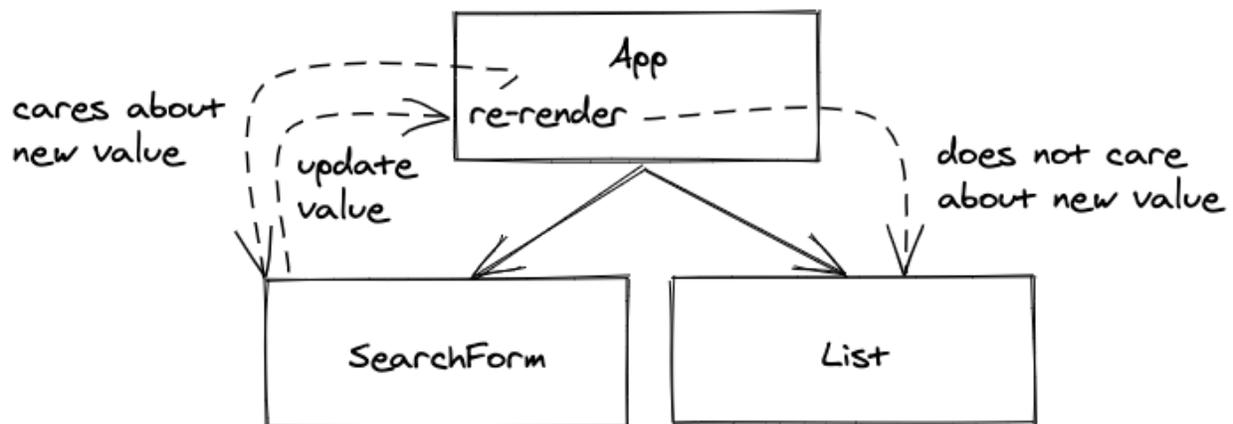
B:List

That's because the `list` passed to the List component is the same, but the `onRemoveItem` callback handler isn't. If the App component re-renders, it always creates a new version of this callback handler as a new function. Earlier, we used React's useCallback Hook to prevent this behavior, by creating a function only on a re-render (if one of its dependencies has changed):

src/App.js

```
const App = () => {  
  ...  
  
  const handleRemoveStory = React.useCallback((item) => {  
    dispatchStories({  
      type: 'REMOVE_STORY',  
      payload: item,  
    });  
  }, []);  
  
  ...  
  
  console.log('B:App');  
  
  return (... );  
};
```

Since the callback handler gets the `item` passed as an argument in its function signature, it doesn't have any dependencies and is declared only once when the App component initially renders. None of the props passed to the List component should change now. Try it with the combination of React `memo` and `useCallback`, to search via the SearchForm's input field. The "B:List" logging disappears, and only the App component re-renders with its "B:App" logging.



While all props passed to a component stay the same, the component renders again if its parent component is forced to re-render. That's React's default behavior, which works most of the time because the re-rendering mechanism is pretty fast. However, if re-rendering decreases the performance of a React application, React's `memo` API helps prevent re-rendering. As we have seen, sometimes `memo` alone doesn't help, though. Callback handlers are re-defined each time in the parent component and passed as *changed* props to the component, which causes another re-render. In

that case, `useCallback` is used for making the callback handler only change when its dependencies change.

Exercises:

- Read more about [React's memo API²⁴²](#).
- Read more about [React's useCallback Hook²⁴³](#).

Don't rerun expensive computations

Sometimes we'll have performance-intensive computations in our React components – between a component's function signature and return block – which run on every render. For this scenario, we must create a use case in our current application first:

src/App.js

```
const getSumComments = (stories) => {
  console.log('C');

  return stories.data.reduce(
    (result, value) => result + value.num_comments,
    0
  );
};

const App = () => {
  ...

  const sumComments = getSumComments(stories);

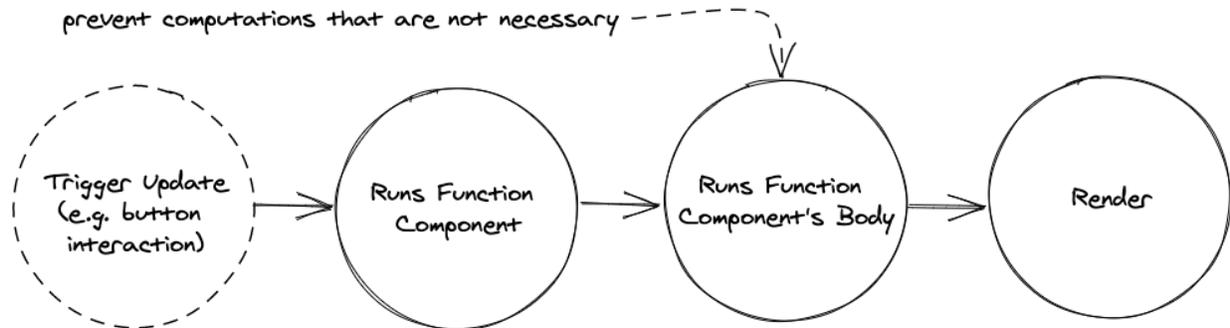
  return (
    <div>
      <h1>My Hacker Stories with {sumComments} comments.</h1>
      ...
    </div>
  );
};
```

If all arguments are passed to a function, it's acceptable to have it outside the component, because it does not have any further dependency needed from within the component. This prevents creating

²⁴²<https://www.robinwieruch.de/react-memo>

²⁴³<https://www.robinwieruch.de/react-usecallback-hook>

the function on every render, so the `useCallback` hook becomes unnecessary. However, the function still computes the value of summed comments on every render, which becomes a problem for more expensive computations.



Each time text is typed in the input field of the SearchForm component, this computation runs again with an output of “C”. This may be fine for a non-heavy computation like this one, but imagine this computation would take more than 500ms. It would give the re-rendering a delay, because everything in the component has to wait for this computation. We can tell React to only run a function if one of its dependencies has changed. If no dependency changed, the result of the function stays the same. React’s `useMemo` Hook helps us here:

`src/App.js`

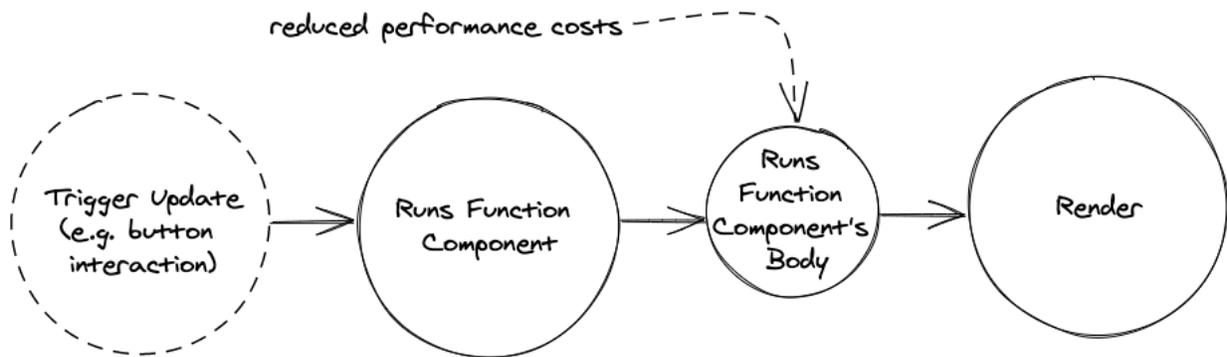
```

const App = () => {
  ...

  const sumComments = React.useMemo(() => getSumComments(stories), [
    stories,
  ]);

  return ( ... );
};
  
```

For every time someone types in the SearchForm, the computation shouldn’t run again. It only runs if the dependency array, here `stories`, has changed. After all, this should only be used for cost expensive computations which could lead to a delay of a (re-)rendering of a component.



Now, after we went through these scenarios for `useMemo`, `useCallback`, and `memo`, remember that these shouldn't necessarily be used by default. Apply these performance optimizations only if you run into performance bottlenecks. Most of the time this shouldn't happen, because React's rendering mechanism is pretty efficient by default. Sometimes the check for utilities like `memo` can be more expensive than the re-rendering itself.

Exercises:

- Confirm your [source code](#)²⁴⁴.
 - Confirm the [changes](#)²⁴⁵.
- Read more about [React's useMemo Hook](#)²⁴⁶.
- Download *React Developer Tools* as an extension for your browser. Open it for your application in the browser via the browser's developer tools and try its various features. For example, you can use it to visualize React's component tree and its updating components.
- Does the SearchForm re-render when removing an item from the List with the "Dismiss" button? If it's the case, apply performance optimization techniques to prevent re-rendering.
- Does each Item re-render when removing an item from the List with the "Dismiss" button? If it's the case, apply performance optimization techniques to prevent re-rendering.
- Remove all performance optimizations to keep the application simple. Our current application doesn't suffer from any performance bottlenecks. Try to avoid [premature optimzations](#)²⁴⁷. Use this section and its further reading material as a reference, in case you run into performance problems.
- Optional: [Rate this section](#)²⁴⁸.

²⁴⁴<https://bit.ly/3AYYPj>

²⁴⁵<https://bit.ly/3aVxXgc>

²⁴⁶<https://www.robinwieruch.de/react-usememo-hook>

²⁴⁷<https://bit.ly/3AYktL8>

²⁴⁸<https://forms.gle/FwNrJsdLikquVzsB6>

TypeScript in React

TypeScript for JavaScript and React has many benefits for developing robust applications. Instead of getting type errors on runtime in the command line or browser, TypeScript integration presents them during compile time inside the IDE. It shortens the feedback loop for a developer, while it improves the developer experience. The code becomes more self-documenting and readable, because every variable is defined with a type. Also moving code blocks or performing a larger refactoring of a codebase becomes much more efficient. Statically typed languages like TypeScript are trending because of their benefits over dynamically typed languages like JavaScript. It's useful to learn more [about TypeScript²⁴⁹](#) whenever possible.

To use TypeScript in React, install TypeScript and its dependencies into your application using the command line. If you run into obstacles, follow the official TypeScript installation instructions for [create-react-app²⁵⁰](#):

Command Line

```
npm install --save typescript @types/node @types/react
npm install --save typescript @types/react-dom @types/jest
```

Next, rename all JavaScript files (*.js*) to TypeScript files (*.tsx*).

Command Line

```
mv src/index.js src/index.tsx
mv src/App.js src/App.tsx
```

Restart your development server in the command line. You may encounter compile errors in the browser and IDE. If the latter doesn't work, try installing a TypeScript plugin for your editor, or extension for your IDE. After the initial TypeScript in React setup, we'll add [type safety²⁵¹](#) for the entire *src/App.tsx* file, starting with typing the arguments of the custom hook:

²⁴⁹<https://bit.ly/3G0l3vL>

²⁵⁰<https://bit.ly/3DYG880>

²⁵¹<https://bit.ly/3jhm6xi>

src/App.tsx

```
const useSemiPersistentState = (  
  key: string,  
  initialState: string  
) => {  
  const [value, setValue] = React.useState(  
    localStorage.getItem(key) || initialState  
  );  
  
  React.useEffect(() => {  
    localStorage.setItem(key, value);  
  }, [value, key]);  
  
  return [value, setValue];  
};
```

Adding types to the function's arguments is more about Javascript than React. We are telling the function to expect two arguments, which are JavaScript string primitives. Also, we can tell the function to return an array ([]) with a string (state), and tell functions like the state updater function that takes a value to return nothing (void):

src/App.tsx

```
const useSemiPersistentState = (  
  key: string,  
  initialState: string  
): [string, (newValue: string) => void] => {  
  const [value, setValue] = React.useState(  
    localStorage.getItem(key) || initialState  
  );  
  
  React.useEffect(() => {  
    localStorage.setItem(key, value);  
  }, [value, key]);  
  
  return [value, setValue];  
};
```

Related to React though, considering the previous type safety improvements for the custom hook, we hadn't to add types to the internal React hooks in the function's body. That's because **type inference** works most of the time for React hooks out of the box. If the *initial state* of a React useState Hook is a JavaScript string primitive, then the returned *current state* will be inferred as a string and the returned *state updater function* will only take a string as an argument and return nothing:

Code Playground

```
const [value, setValue] = React.useState('React');  
// value is inferred to be a string  
// setValue only takes a string as argument
```

If adding type safety becomes an aftermath for a React application and its components, there are multiple ways on how to approach it. We will start with the props and state for the leaf components of our application. For example, the `Item` component receives a story (here `item`) and a callback handler function (here `onRemoveItem`). Starting out very verbose, we could add the inlined types for both function arguments as we did before:

src/App.tsx

```
const Item = ({  
  item,  
  onRemoveItem,  
}: {  
  item: {  
    objectID: string;  
    url: string;  
    title: string;  
    author: string;  
    num_comments: number;  
    points: number;  
  };  
  onRemoveItem: (item: {  
    objectID: string;  
    url: string;  
    title: string;  
    author: string;  
    num_comments: number;  
    points: number;  
  }) => void;  
}) => (  
  <li>  
    ...  
  </li>  
)
```

There are two problems: the code is verbose, and it has duplicates. Let's get rid of both problems by defining a custom `Story` type outside the component, at the top of `src/App.js`:

src/App.tsx

```
type Story = {
  objectID: string;
  url: string;
  title: string;
  author: string;
  num_comments: number;
  points: number;
};

...

const Item = ({
  item,
  onRemoveItem,
}: {
  item: Story;
  onRemoveItem: (item: Story) => void;
}) => (
  <li>
    ...
  </li>
);
```

The `item` is of type `Story`; the `onRemoveItem` function takes an `item` of type `Story` as an argument and returns nothing. Next, clean up the code by defining the props of the `Item` component outside:

src/App.tsx

```
type ItemProps = {
  item: Story;
  onRemoveItem: (item: Story) => void;
};

const Item = ({ item, onRemoveItem }: ItemProps) => (
  <li>
    ...
  </li>
);
```

That's the most popular way to type React component's props with TypeScript. Fortunately, the return type of the function component is inferred. However, if you want to explicitly use it, you can do so with `JSX.Element`. From here, we can navigate up the component tree into the `List` component and apply the same type definitions for the props:

src/App.tsx

```
type Story = {
  ...
};

type Stories = Array<Story>;

...

type ListProps = {
  list: Stories;
  onRemoveItem: (item: Story) => void;
};

const List = ({ list, onRemoveItem }: ListProps) => (
  <ul>
    {list.map((item) => (
      <Item
        key={item.objectID}
        item={item}
        onRemoveItem={onRemoveItem}
      />
    ))}
  </ul>
);
```

The `onRemoveItem` function is typed twice for the `ItemProps` and `ListProps`. To be more accurate, you *could* extract this to a standalone defined `OnRemoveItem` TypeScript type and reuse it for both `onRemoveItem` prop type definitions. Note, however, that development becomes increasingly difficult as components are split up into different files. That's why we will keep the duplication here. Now, since we already have the `Story` and `Stories` types, we can repurpose them for other components. Add the `Story` type to the callback handler in the `App` component:

src/App.tsx

```
const App = () => {  
  ...  
  
  const handleRemoveStory = (item: Story) => {  
    dispatchStories({  
      type: 'REMOVE_STORY',  
      payload: item,  
    });  
  };  
  
  ...  
};
```

The reducer function manages the `Story` type as well, without really touching it due to `state` and `action` types. As the application's developer, we know both objects and their shapes that are passed to this reducer function:

src/App.tsx

```
type StoriesState = {  
  data: Stories;  
  isLoading: boolean;  
  isError: boolean;  
};  
  
type StoriesAction = {  
  type: string;  
  payload: any;  
};  
  
const storiesReducer = (  
  state: StoriesState,  
  action: StoriesAction  
) => {  
  ...  
};
```

The `Action` type with its `string` and `any` (TypeScript **wildcard**) type definitions are still too broad; and we gain no real type safety through it, because actions are not distinguishable. We can do better by specifying each action TypeScript type as an **interface**, and using a **union type** (here `StoriesAction`) for the final type safety:

src/App.tsx

```
interface StoriesFetchInitAction {
  type: 'STORIES_FETCH_INIT';
}

interface StoriesFetchSuccessAction {
  type: 'STORIES_FETCH_SUCCESS';
  payload: Stories;
}

interface StoriesFetchFailureAction {
  type: 'STORIES_FETCH_FAILURE';
}

interface StoriesRemoveAction {
  type: 'REMOVE_STORY';
  payload: Story;
}

type StoriesAction =
  | StoriesFetchInitAction
  | StoriesFetchSuccessAction
  | StoriesFetchFailureAction
  | StoriesRemoveAction;

const storiesReducer = (
  state: StoriesState,
  action: StoriesAction
) => {
  ...
};
```

The reducer's current state, action, and returned state (inferred) are type safe now. For example, if you would dispatch an action to the reducer with an action type that's not defined, you would get a type error. Or if you would pass something else than a story to the action which removes a story, you would get a type error as well.

Let's shift our focus to the SearchForm component, which has callback handlers with events:

src/App.tsx

```
type SearchFormProps = {
  searchTerm: string;
  onSearchInput: (event: React.ChangeEvent<HTMLInputElement>) => void;
  onSearchSubmit: (event: React.FormEvent<HTMLFormElement>) => void;
};

const SearchForm = ({
  searchTerm,
  onSearchInput,
  onSearchSubmit,
}: SearchFormProps) => (
  ...
);
```

Often using `React.SyntheticEvent` instead of `React.ChangeEvent` or `React.FormEvent` is usually enough. Going up to the App component again, we apply the same type for the callback handler there:

src/App.tsx

```
const App = () => {
  ...

  const handleSearchInput = (
    event: React.ChangeEvent<HTMLInputElement>
  ) => {
    setSearchTerm(event.target.value);
  };

  const handleSearchSubmit = (
    event: React.FormEvent<HTMLFormElement>
  ) => {
    setUrl(`${API_ENDPOINT}${searchTerm}`);

    event.preventDefault();
  };

  ...
};
```

All that's left is the `InputWithLabel` component. Before handling this component's props, let's take a look at the `ref` from React's `useRef` Hook. Unfortunately, the return value isn't inferred:

src/App.tsx

```
const InputWithLabel = ({ ... }) => {
  const inputRef = React.useRef<HTMLInputElement>(null!);

  React.useEffect(() => {
    if (isFocused && inputRef.current) {
      inputRef.current.focus();
    }
  }, [isFocused]);
```

We made the returned ref type safe, and typed it as read-only because we only execute the focus method on it (read). React takes over for us there, setting the DOM element to the current property.

Lastly, we will apply type safety checks for the InputWithLabel component's props. Note the children prop with its React specific type and the **optional types** are signaled with a question mark:

src/App.tsx

```
type InputWithLabelProps = {
  id: string;
  value: string;
  type?: string;
  onChange: (event: React.ChangeEvent<HTMLInputElement>) => void;
  isFocused?: boolean;
  children: React.ReactNode;
};

const InputWithLabel = ({
  id,
  value,
  type = 'text',
  onChange,
  isFocused,
  children,
}: InputWithLabelProps) => {
  ...
};
```

Both the type and isFocused properties are optional. Using TypeScript, you can tell the compiler that these don't need to be passed to the component as props. The children prop has a lot of TypeScript type definitions that could be applicable to this concept, the most universal of which is React.ReactNode from the React library.

Our entire React application is finally typed by TypeScript, making it easy to spot type errors on compile time. When adding TypeScript to your React application, start by adding type definitions to your function's arguments. These functions can be vanilla JavaScript functions, custom React hooks, or React function components. Only when using React is it important to know specific types for form elements, events, and JSX.

Exercises:

- Confirm your [source code](#)²⁵².
 - Confirm the [changes](#)²⁵³.
- Dig into the [React + TypeScript Cheatsheet](#)²⁵⁴, because most common use cases we faced in this section are covered there as well. There is no need to know everything from the top of your head.
- While you continue with the learning experience in the following sections, remove or keep your types with TypeScript. If you do the latter, add new types whenever you get a compile error.
- Optional: [Rate this section](#)²⁵⁵.

²⁵²<https://bit.ly/3vtF8p4>

²⁵³<https://bit.ly/3pjxlJr>

²⁵⁴<https://bit.ly/3phdf2H>

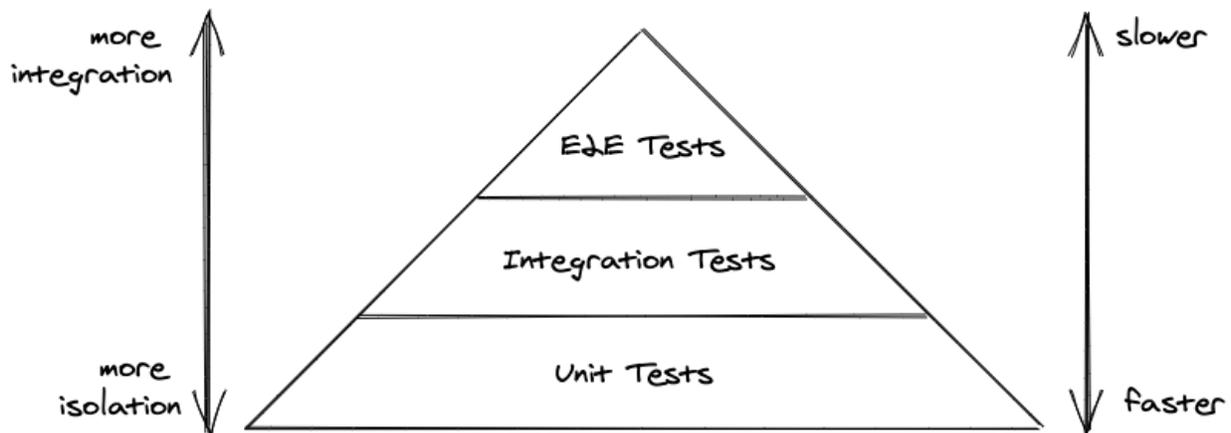
²⁵⁵<https://forms.gle/Pyw2oUjXV85hwk2t6>

Testing in React

Testing source code is an essential part of programming and should be seen as a mandatory exercise for serious developers. The goal is to verify our source code's quality and functionality before using it in production. The [testing pyramid](#)²⁵⁶ will serve as our guide.

The testing pyramid includes end-to-end tests, integration tests, and unit tests. Unit tests are for small, isolated blocks of code, such as a single function or component. Integration tests help us figure out how well these blocks of code work together. An end-to-end test simulates a real-life scenario, like a user logging into a web application. Unit tests are quick and easy to write and maintain; end-to-end tests are the opposite.

Many unit tests are required to cover all the functions and components in a working application, after which several integration tests make sure that the most important units work together. Finally, a few end-to-end tests to simulate critical user scenarios. In this learning experience, we will cover **unit and integration tests**, in addition to a useful component-specific testing technique called **snapshot tests**. E2E tests will be part of the exercise.



Choosing a testing library can be a challenge for React beginners, as there are many options. To keep things simple, we'll employ the most popular tools: [Jest](#)²⁵⁷ and [React Testing Library](#)²⁵⁸ (RTL). Jest is a full-blown testing framework with test runners, test suites, test cases, and assertions. RTL is used for rendering React components, triggering events like mouse clicks, and selecting HTML elements from the DOM to perform assertions. We'll explore both tools step-by-step, from setup to unit testing to integration testing.

Test Suites, Test Cases, and Assertions

Test suites and test cases are commonly used in JavaScript and many other programming languages. A test suite groups the individual test cases into one larger subject. Let's see how this looks with Jest

²⁵⁶<https://bit.ly/3BYEra1>

²⁵⁷<https://jestjs.io>

²⁵⁸<https://testing-library.com>

in our `src/App.test.js` file:

`src/App.test.js`

```
describe('something truthy and falsy', () => {
  test('true to be true', () => {
    expect(true).toBe(true);
  });

  test('false to be false', () => {
    expect(false).toBe(false);
  });
});
```

The “describe” block is our *test suite*, and the “test” blocks are our *test cases*. Note that test cases can be used without test suites, which may apply to code outside the scope of this lesson:

`src/App.test.js`

```
test('true to be true', () => {
  expect(true).toBe(true);
});

test('false to be false', () => {
  expect(false).toBe(false);
});
```

Large subjects like functions or components often require multiple test cases, so it makes sense to use them with test suites:

`src/App.test.js`

```
describe('App component', () => {
  test('removes an item when clicking the Dismiss button', () => {

  });

  test('requests some initial stories from an API', () => {

  });
});
```

Note that a “test” block can also be written as an “it” block. The blocks have the same purpose, except the “it” block may be more familiar to programmers from other programming languages:

src/App.test.js

```
describe('something truthy and falsy', () => {
  it('true to be true', () => {
    expect(true).toBe(true);
  });

  it('false to be false', () => {
    expect(false).toBe(false);
  });
});
```

Fortunately, create-react-app comes with Jest, so you don't need to install anything. You can run tests using the test script from your *package.json* in the command line as soon as create-react-app is installed, and then execute your tests with `npm test` to produce the following output:

Command Line

```
PASS  src/App.test.js
  something truthy and falsy
    ✓ true to be true (3ms)
    ✓ false to be false (1ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        2.78s, estimated 4s
Ran all test suites related to changed files.
```

Watch Usage

- › Press a to run all tests.
 - › Press f to run only failed tests.
 - › Press q to quit watch mode.
 - › Press p to filter by a filename regex pattern.
 - › Press t to filter by a test name regex pattern.
 - › Press Enter to trigger a test run.
-

When we run the test command, the test runner matches all files with a *test.js* suffix. Successful tests are displayed in green, failed tests in red. The interactive test script watches your tests and source code and executes tests when the files change. Jest also provides a few interactive commands, such as pressing “f” to run failed tests and “a” for running all tests. Let's see how this looks for a failed test:

src/App.test.js

```
describe('something truthy and falsy', () => {
  test('true to be true', () => {
    expect(true).toBe(true);
  });

  test('false to be false', () => {
    expect(false).toBe(true);
  });
});
```

The tests run again, and the command line output shows a failed test in red:

Command Line

```
FAIL src/App.test.js
something truthy and falsy
  ✓ true to be true (2ms)
  ✖ false to be false (4ms)

  ✖ something truthy and falsy > false to be false

    expect(received).toBe(expected) // Object.is equality

    Expected: true
    Received: false

       5 |
       6 |   test('false to be false', () => {
     >  7 |     expect(false).toBe(true);
         |                               ^
       8 |   });
       9 | });
      10 |

      at Object.<anonymous> (src/App.test.js:7:19)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 passed, 2 total
Snapshots:  0 total
Time:        3.385s
Ran all test suites related to changed files.
```

Watch Usage: Press w to show more.

Familiarize yourself with this test output, because it shows all failed tests, as well as information on why they failed. Using this information, you can fix certain parts of your code until all tests run green. Next, we'll cover *test assertions*, two of which we've already used with Jest's `expect` function. An assertion works by expecting value on the left side (`expect`) to match a value on the right side (`toBe`). `toBe` is only one of many available assertive functions provided by Jest.

`src/App.test.js`

```
describe('something truthful and falsy', () => {
  test('true to be true', () => {
    expect(true).toBeTruthy();
  });

  test('false to be false', () => {
    expect(false).toBeFalsy();
  });
});
```

Once you start testing, it's a good practice to keep two command line interfaces open: one for watching your tests (`npm test`), and one for developing your application (`npm start`). Also, source control platforms like Git may require an additional command line interface for adding your source code to the repository.

Exercises:

- Confirm your [source code](#)²⁵⁹.
 - Confirm the [changes](#)²⁶⁰.
- Read more about [Jest](#)²⁶¹.

Unit Testing: Functions

A unit test is generally used to test components or functions in isolation. For functions, unit tests are for input and output; for components, we test props or the callback handlers communicating to the outside. Before we can perform a unit test on our `src/App.js` file, we must export components and functions like the reducer from our `src/App.js` file with a named export:

²⁵⁹<https://bit.ly/3C42rII>

²⁶⁰<https://bit.ly/3B2CesG>

²⁶¹<https://jestjs.io>

src/App.js

...

```
export default App;
```

```
export { storiesReducer, SearchForm, InputWithLabel, List, Item };
```

The exercises at the end of this chapter will cover all the remaining tests you should consider performing. For now, we can import all the components and reducers in our *src/App.test.js* file and we will focus on the reducer test first. We are also importing React here, because we have to include it whenever we test React components:

src/App.test.js

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

```
import App, {  
  storiesReducer,  
  Item,  
  List,  
  SearchForm,  
  InputWithLabel,  
} from './App';
```

Before we unit test our first React component, we'll cover how to test just a JavaScript function. The best candidate for this test use case is the *storiesReducer* function and one of its actions. Let's define some test data and the test suite for the reducer test:

src/App.test.js

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

...

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

```
const storyTwo = {
  title: 'Redux',
  url: 'https://redux.js.org/',
  author: 'Dan Abramov, Andrew Clark',
  num_comments: 2,
  points: 5,
  objectID: 1,
};

const stories = [storyOne, storyTwo];

describe('storiesReducer', () => {
  test('removes a story from all stories', () => {

  });
});
```

If you extrapolate the test cases, there should be one test case per reducer action. We will focus on a single action, which you can use to perform the rest as exercise yourself. The reducer function accepts a state and an action, and then returns a new state, so reducer tests follow the same pattern:

src/App.test.js

```
...

describe('storiesReducer', () => {
  test('removes a story from all stories', () => {
    const action = // TODO: some action
    const state = // TODO: some current state

    const newState = storiesReducer(state, action);

    const expectedState = // TODO: the expected state

    expect(newState).toBe(expectedState);
  });
});
```

For our case, we define action, state, and expected state according to our reducer. The expected state will have one less story, which was removed as it passed to the reducer as action:

src/App.test.js

```
describe('storiesReducer', () => {
  test('removes a story from all stories', () => {
    const action = { type: 'REMOVE_STORY', payload: storyOne };
    const state = { data: stories, isLoading: false, isError: false };

    const newState = storiesReducer(state, action);

    const expectedState = {
      data: [storyTwo],
      isLoading: false,
      isError: false,
    };

    expect(newState).toBe(expectedState);
  });
});
```

This test still fails because we are using `toBe` instead of `toStrictEqual`. The `toBe` assertive function makes a strict comparison like `newState === expectedState`. The content of the objects are the same, however, their object references are not the same. We use `toStrictEqual` instead of `toBe` to limit our comparison to the object's content:

src/App.test.js

```
describe('storiesReducer', () => {
  test('removes a story from all stories', () => {
    const action = { type: 'REMOVE_STORY', payload: storyOne };
    const state = { data: stories, isLoading: false, isError: false };

    const newState = storiesReducer(state, action);

    const expectedState = {
      data: [storyTwo],
      isLoading: false,
      isError: false,
    };

    expect(newState).toStrictEqual(expectedState);
  });
});
```

There is always the decision to make for JavaScript objects whether you want to make a strict comparison or just a content comparison. Most often you only want to have a content comparison

here, hence use `toStrictEqual`. For JavaScript primitives though, like strings or booleans, you can still use `toBe`. Also note that there is a `toEqual` function [which works slightly different](#)²⁶² than `toStrictEqual`.

We continue to make adjustments until the reducer test turns green, which is really testing a JavaScript function with a certain input and expecting a certain output. We haven't done any testing methods regarding React yet.

Remember, a reducer function will always follow the same test pattern: given a state and action, we expect the following new state. Every action of the reducer could be another test case in our reducer's test suite, so consider using the exercises as a way to move through your entire source code.

Exercises:

- Confirm your [source code](#)²⁶³.
 - Confirm the [changes](#)²⁶⁴.
- Continue to write a test case for every reducer action and its state transition.
- Read more about Jest's assertive functions like `toBe` and `toStrictEqual`.

Unit Testing: Components

We tested our first function in JavaScript with Jest in the previous sections. Next, we'll test our first isolated React component with a unit test. Since we are using create-react-app, we don't need to set up the React Testing Library (RTL), which is necessary for our component tests because it is the default testing library. If you are using a custom React setup like React with Webpack, RTL as a library would need to be installed.

The following functions from React Testing Library are used for component tests:

```
src/App.test.js
```

```
import * as React from 'react';
import {
  render,
  screen,
  fireEvent,
  act,
} from '@testing-library/react';
```

```
...
```

²⁶²<https://bit.ly/3jIPpii>

²⁶³<https://bit.ly/3jllhD3>

²⁶⁴<https://bit.ly/3E175aO>

Start with the `Item` component, where we assert whether it renders all expected properties based on its given props. Based on the input (props), we are asserting an output (rendered HTML). We'll use RTL's `render` function in each test to render a React component. In this case, we render the `Item` component as an element and pass it an `item` object – one of our previously defined stories – as props:

`src/App.test.js`

```
describe('Item', () => {
  test('renders all properties', () => {
    render(<Item item={storyOne} />);
  });
});
```

After rendering it, we can use the `debug` function from RTL's `screen` object:

`src/App.test.js`

```
describe('Item', () => {
  test('renders all properties', () => {
    render(<Item item={storyOne} />);

    screen.debug();
  });
});
```

Run the tests with `npm test`, and you'll see the output from the `debug` function. It prints all your component's and child component's HTML elements. The output should be similar to the following:

`src/App.test.js`

```
<body>
  <div>
    <li>
      <span>
        <a
          href="https://reactjs.org/"
        >
          React
        </a>
      </span>
      <span>
        Jordan Walke
      </span>
      <span>
```

```
    3
    </span>
    <span>
      4
    </span>
    <span>
      <button
        type="button"
      >
        Dismiss
      </button>
    </span>
  </li>
</div>
</body>
```

Here you should form the habit of using RTL's debug function whenever you render a new component in a React component test. The function gives a useful overview of what is rendered and informs the best way to proceed with testing. Based on the current output, we can start with our first assertion. RTL's screen object provides a function called `getByText`, one of many search functions:

`src/App.test.js`

```
describe('Item', () => {
  test('renders all properties', () => {
    render(<Item item={storyOne} />);

    expect(screen.getByText('Jordan Walke')).toBeInTheDocument();
    expect(screen.getByText('React')).toHaveAttribute(
      'href',
      'https://reactjs.org/'
    );
  });
});
```

For the two assertions, we use the two assertive functions `toBeInTheDocument` and `toHaveAttribute`. These are to verify an element with the text “Jordan Walke” is in the document, and the presence of an element with the text “React” with a specific `href` attribute value. Over time, you will see more of these assertive functions being used.

RTL's `getByText` search function finds the one element with the visible texts “Jordan Walke” and “React”. We can use the `getAllByText` equivalent to find more than one element. Similar equivalents exist for other search functions.

The `getByText` function returns the element with a text that users can see, which relates to the real-world use of the application. Note that `getByText` is not the only search function, though. Another highly-used search function is the `getByRole` or `getAllByRole` function:

src/App.test.js

```
describe('Item', () => {
  test('renders all properties', () => {
    ...
  });

  test('renders a clickable dismiss button', () => {
    render(<Item item={storyOne} />);

    expect(screen.getByRole('button')).toBeInTheDocument();
  });
});
```

The `getByRole` function is usually used to retrieve elements by [aria-label attributes](#)²⁶⁵. However, there are also [implicit roles on HTML elements](#)²⁶⁶ – like `button` for a button element. Thus you can select elements not only by visible text, but also by their (implicit) accessibility role with React Testing Library. A neat feature of `getRoleBy` is that [it suggests roles if you provide a role that's not available](#)²⁶⁷. Both, `getByText` and `getByRole` are RTL's most widely used search functions.

We can continue here by asserting not only that everything is *in the document*, but also by asserting whether our events work as expected. For example, the `Item` component's button element can be clicked and we want to verify that the callback handler gets called. Therefore, we are using Jest for creating a mocked function which we provide as a callback handler to the `Item` component. Then, after firing a click event with React Testing Library on the button, we want to assert that the callback handler function has been called:

src/App.test.js

```
describe('Item', () => {
  test('renders all properties', () => {
    ...
  });

  test('renders a clickable dismiss button', () => {
    ...
  });
});
```

²⁶⁵<https://mzl.la/3B3bBDP>

²⁶⁶<https://mzl.la/3n7SgN7>

²⁶⁷<https://bit.ly/3pnPXrQ>

```
test('clicking the dismiss button calls the callback handler', () => {
  const handleRemoveItem = jest.fn();

  render(<Item item={storyOne} onRemoveItem={handleRemoveItem} />);

  fireEvent.click(screen.getByRole('button'));

  expect(handleRemoveItem).toHaveBeenCalledTimes(1);
});
```

Jest lets us pass a test-specific function to the Item component as a prop. These test-specific functions are called **spy**, **stub**, or **mock**; each is used for different test scenarios. The `jest.fn()` returns us a *mock* for the actual function, which lets us capture when it's called. As a result, we can use Jest assertions like `toHaveBeenCalledTimes`, which lets us assert a number of times the function has been called; and `toHaveBeenCalledWith`, to verify arguments that are passed to it.

Every time we want to spy a JavaScript function, whether it has been called or whether it received certain arguments, we can use Jest's helper function to create a mocked function. Then, after invoking this function implicitly with RTL's `fireEvent` object's function, we can assert that the provided callback handler – which is the mocked function – has been called one time.

In the last exercise we tested the Item component's input and output via rendering assertions and callback handler assertions. We are not testing real state changes yet, however, as there is no actual item removed from the DOM after clicking the “Dismiss” button. The logic to remove the item from the list is in the App component, but we are only testing the Item component in isolation. Sometimes it's just useful to test whether a single block works, before testing everything all together. We will test the actual implementation logic for removing an Item when we cover the App component later.

For now, the SearchForm component will use the InputWithLabel component as a child component. As before, we will start by rendering the component and providing all the essential props:

src/App.test.js

```
describe('SearchForm', () => {
  const searchFormProps = {
    searchTerm: 'React',
    onSearchInput: jest.fn(),
    onSearchSubmit: jest.fn(),
  };

  test('renders the input field with its value', () => {
    render(<SearchForm {...searchFormProps} />);

    screen.debug();
```

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

Again, we start with the debugging. After evaluating what renders, we can make the first assertion for the SearchForm. With input fields, the `getByDisplayValue` search function is the perfect candidate to return the input field as an element:

src/App.test.js

```
describe('SearchForm', () => {  
  const searchFormProps = { ... };  
  
  test('renders the input field with its value', () => {  
    render(<SearchForm {...searchFormProps} />);  
  
    expect(screen.getByDisplayValue('React')).toBeInTheDocument();  
  });  
});
```

Since the input element is rendered with a default value, we can use the default value (here “React”), which is the displayed value in our test assertion. If the input element doesn’t have a default value, the application could show a placeholder with the `placeholder` HTML attribute on the input field. Then we’d use another function from RTL called `getByPlaceholderText`, which is used for searching an element with a placeholder text.

Because the debug information presented multiple options to query the HTML, we could continue with one more test to assert the rendered label:

src/App.test.js

```
describe('SearchForm', () => {  
  const searchFormProps = { ... };  
  
  test('renders the input field with its value', () => {  
    ...  
  });  
  
  test('renders the correct label', () => {  
    render(<SearchForm {...searchFormProps} />);  
  
    expect(screen.getByLabelText(/Search/)).toBeInTheDocument();  
  });  
});
```

The `getByLabelText` search function allows us to find an element by a label in a form. This is useful for components that render multiple labels and HTML controls. However, you may have noticed we used a [regular expression](#)²⁶⁸ here. If we used a string instead, the colon for “Search:” must be included. By using a regular expression, we are matching strings that include the “Search” string, which makes finding elements much more efficient. For this reason, you may find yourself using regular expressions instead of strings quite often.

Anyway, perhaps it would be more interesting to test the interactive parts of the `SearchForm` component. Since our callback handlers, which are passed as props to the `SearchForm` component, are already mocked with Jest, we can assert whether these functions get called appropriately:

`src/App.test.js`

```
describe('SearchForm', () => {
  const searchFormProps = {
    searchTerm: 'React',
    onSearchInput: jest.fn(),
    onSearchSubmit: jest.fn(),
  };

  ...

  test('calls onSearchInput on input field change', () => {
    render(<SearchForm {...searchFormProps} />);

    fireEvent.change(screen.getByDisplayValue('React'), {
      target: { value: 'Redux' },
    });

    expect(searchFormProps.onSearchInput).toHaveBeenCalledTimes(1);
  });

  test('calls onSearchSubmit on button submit click', () => {
    render(<SearchForm {...searchFormProps} />);

    fireEvent.submit(screen.getByRole('button'));

    expect(searchFormProps.onSearchSubmit).toHaveBeenCalledTimes(1);
  });
});
```

Similar to the `Item` component, we tested input (props) and output (callback handler) for the `SearchForm` component. The difference is that the `SearchForm` component renders a child component called

²⁶⁸<https://mzl.la/3CdDjiZ>

`InputWithLabel`. If you check the debug output, you'll likely notice that the React Testing Library doesn't bother with this child component. This happens because the end-user wouldn't care about the component either, so the React Testing Library outputs all the HTML that matters for the test.

All the callback handler tests for `Item` and `SearchForm` component test only whether the functions have been called. No React re-rendering occurs, because all the components are tested in isolation without state management, which solely happens in the `App` component. Real testing with RTL starts further up the component tree, where state changes and side-effects can be evaluated. Therefore, let me introduce integration testing next.

Exercises:

- Confirm your [source code](#)²⁶⁹.
 - Confirm the [changes](#)²⁷⁰.
- Read more about [React Testing Library](#)²⁷¹.
 - Read more about [search functions](#)²⁷².
- Add tests for your `List` and `InputWithLabel` components.

Integration Testing: Component

React Testing Library adheres to a single core philosophy: instead of testing implementation details of React components, it tests how users interact with the application and if it works as expected. This becomes especially powerful for integration tests.

We'll need to provide some data before we test the `App` component, since it makes requests for data from the remote API after its initial render. We'll start by using `axios` in the `App` component for our data request, after which we'll mock it with `Jest` at the top of the testing file:

```
src/App.test.js
```

```
...
```

```
import axios from 'axios';
```

```
...
```

```
jest.mock('axios');
```

```
...
```

Next, implement the data you want to be returned from the mocked API request with a JavaScript Promise, and use it for the `axios` mock. Afterward, we can render our component and assume the correct data is mocked for our API request:

²⁶⁹<https://bit.ly/2Z3Vc54>

²⁷⁰<https://bit.ly/3E28IoZ>

²⁷¹<https://bit.ly/30KueQH>

²⁷²<https://bit.ly/3jjUw2t>

src/App.test.js

```
describe('App', () => {
  test('succeeds fetching data', () => {
    const promise = Promise.resolve({
      data: {
        hits: stories,
      },
    });

    axios.get.mockImplementationOnce(() => promise);

    render(<App />);

    screen.debug();
  });
});
```

Now we'll use React Testing Library's `act` helper function to wait until the promise resolves after the component's initial render. With `async/await`, we can implement this like synchronous code. The `debug` function from RTL is useful because it outputs the App component's elements before and after the request:

src/App.test.js

```
describe('App', () => {
  test('succeeds fetching data', async () => {
    const promise = Promise.resolve({
      data: {
        hits: stories,
      },
    });

    axios.get.mockImplementationOnce(() => promise);

    render(<App />);

    screen.debug();

    await act(() => promise);

    screen.debug();
  });
});
```

In the debug's output, we see the loading indicator renders for the first debug function, but not the second. This is because the data fetching and component re-render complete after we resolve the promise in our test with `act`. Let's assert the loading indicator for this case:

`src/App.test.js`

```
describe('App', () => {
  test('succeeds fetching data', async () => {
    const promise = Promise.resolve({
      data: {
        hits: stories,
      },
    });

    axios.get.mockImplementationOnce(() => promise);

    render(<App />);

    expect(screen.queryByText(/Loading/)).toBeInTheDocument();

    await act(() => promise);

    expect(screen.queryByText(/Loading/)).toBeNull();
  });
});
```

Because we're testing for a returned element that is absent, this time we use RTL's `queryByText` instead of the `getByText` function. Using `getByText` in this instance would produce an error, because the element can't be found; but with `queryByText` the value just returns `null`.

Again, we're using a regular expression `/Loading/` instead of a string `'Loading'`. To use a string, we'd have to explicitly use `'Loading ...'` instead of `'Loading'`. With a regular expression, we don't need to provide the whole string, we just need to match a part of it.

Next, we can assert whether or not our fetched data gets rendered as expected:

src/App.test.js

```
describe('App', () => {
  test('succeeds fetching data', async () => {
    const promise = Promise.resolve({
      data: {
        hits: stories,
      },
    });

    axios.get.mockImplementationOnce(() => promise);

    render(<App />);

    expect(screen.queryByText(/Loading/)).toBeInTheDocument();

    await act(() => promise);

    expect(screen.queryByText(/Loading/)).toBeNull();

    expect(screen.getByText('React')).toBeInTheDocument();
    expect(screen.getByText('Redux')).toBeInTheDocument();
    expect(screen.getAllByText('Dismiss').length).toBe(2);
  });
});
```

The [happy path](#)²⁷³ for the data fetching is tested now. Similarly, we can test the unhappy path in case of a failed API request. The promise needs to reject and the error should be caught with a try/catch block:

src/App.test.js

```
describe('App', () => {
  test('succeeds fetching data', async () => {
    ...
  });

  test('fails fetching data', async () => {
    const promise = Promise.reject();

    axios.get.mockImplementationOnce(() => promise);

    render(<App />);
```

²⁷³<https://bit.ly/3jiAbuB>

```
    expect(screen.getByText(/Loading/)).toBeInTheDocument();

    try {
      await act(() => promise);
    } catch (error) {
      expect(screen.queryByText(/Loading/)).toBeNull();
      expect(screen.queryByText(/went wrong/)).toBeInTheDocument();
    }
  });
});
```

There may be some confusion about when to use `getBy` or the `queryBy` search variants. As a rule of thumb, use `getBy` for single elements, and `getAllBy` for multiple elements. If you are checking for elements that aren't present, use `queryBy` (or `queryAllBy`). In this code, I preferred using `queryBy` for the sake of alignment and readability.

Now we know the initial data fetching works for our `App` component, so we can move to testing user interactions. We have only tested user actions in the child components thus far, by firing events without any state and side-effect. Next, we'll remove an item from the list after the data has been fetched successfully. Since the item with "Jordan Walke" is the first rendered item in the list, it gets removed if we click the first "Dismiss" button:

`src/App.test.js`

```
describe('App', () => {

  ...

  test('removes a story', async () => {
    const promise = Promise.resolve({
      data: {
        hits: stories,
      },
    });

    axios.get.mockImplementationOnce(() => promise);

    render(<App />);

    await act(() => promise);

    expect(screen.getAllByText('Dismiss').length).toBe(2);
    expect(screen.getByText('Jordan Walke')).toBeInTheDocument();
  });
});
```

```
fireEvent.click(screen.getAllByText('Dismiss')[0]);

expect(screen.getAllByText('Dismiss').length).toBe(1);
expect(screen.queryByText('Jordan Walke')).toBeNull();
});
});
```

To test the search feature, we set up the mocking differently, because we're handling initial request, plus another request once the user searches for more stories by a specific search term:

src/App.test.js

```
describe('App', () => {

  ...

  test('searches for specific stories', async () => {
    const reactPromise = Promise.resolve({
      data: {
        hits: stories,
      },
    });

    const anotherStory = {
      title: 'JavaScript',
      url: 'https://en.wikipedia.org/wiki/JavaScript',
      author: 'Brendan Eich',
      num_comments: 15,
      points: 10,
      objectID: 3,
    };

    const javascriptPromise = Promise.resolve({
      data: {
        hits: [anotherStory],
      },
    });

    axios.get.mockImplementation((url) => {
      if (url.includes('React')) {
        return reactPromise;
      }
    });
  });
});
```

```
    if (url.includes('JavaScript')) {
      return javascriptPromise;
    }

    throw Error();
  });
});
});
```

Instead of mocking the request once with Jest (`mockImplementationOnce`), now we mock multiple requests (`mockImplementation`). Depending on the incoming URL, the request either returns the initial list (“React”-related stories), or the new list (“JavaScript”-related stories). If we provide an incorrect URL to the request, the test throws an error for confirmation. As before, let’s render the App component:

`src/App.test.js`

```
describe('App', () => {

  ...

  test('searches for specific stories', async () => {
    const reactPromise = Promise.resolve({ ... });

    const anotherStory = { ... };

    const javascriptPromise = Promise.resolve({ ... });

    axios.get.mockImplementation((url) => {
      ...
    });

    // Initial Render

    render(<App />);

    // First Data Fetching

    await act(() => reactPromise);

    expect(screen.queryByDisplayValue('React')).toBeInTheDocument();
    expect(screen.queryByDisplayValue('JavaScript')).toBeNull();

    expect(screen.queryByText('Jordan Walke')).toBeInTheDocument();
```

```
    expect(
      screen.queryByText('Dan Abramov, Andrew Clark')
    ).toBeInTheDocument();
    expect(screen.queryByText('Brendan Eich')).toBeNull();
  });
});
```

We are resolving the first promise for the initial render. We expect the input field to render “React”, and the two items in the list to render the creators of React and Redux. We also make sure that no stories related to JavaScript are rendered yet. Next, change the input field’s value by firing an event, and asserting that the new value is rendered from the App component through all its child components in the actual input field:

src/App.test.js

```
describe('App', () => {

  ...

  test('searches for specific stories', async () => {

    ...

    expect(screen.queryByText('Jordan Walke')).toBeInTheDocument();
    expect(
      screen.queryByText('Dan Abramov, Andrew Clark')
    ).toBeInTheDocument();
    expect(screen.queryByText('Brendan Eich')).toBeNull();

    // User Interaction -> Search

    fireEvent.change(screen.queryByDisplayValue('React'), {
      target: {
        value: 'JavaScript',
      },
    });

    expect(screen.queryByDisplayValue('React')).toBeNull();
    expect(
      screen.queryByDisplayValue('JavaScript')
    ).toBeInTheDocument();
  });
});
```

Lastly, we can submit this search request by firing a submit event with the button. The new search term is used from the App component's state, so the new URL searches for JavaScript-related stories that we have mocked before:

src/App.test.js

```
describe('App', () => {  
  
  ...  
  
  test('searches for specific stories', async () => {  
  
    ...  
  
    expect(screen.queryByDisplayValue('React')).toBeNull();  
    expect(  
      screen.queryByDisplayValue('JavaScript')  
    ).toBeInTheDocument();  
  
    fireEvent.submit(screen.queryByText('Submit'));  
  
    // Second Data Fetching  
  
    await act(() => javascriptPromise);  
  
    expect(screen.queryByText('Jordan Walke')).toBeNull();  
    expect(  
      screen.queryByText('Dan Abramov, Andrew Clark')  
    ).toBeNull();  
    expect(screen.queryByText('Brendan Eich')).toBeInTheDocument();  
  });  
});
```

Brendan Eich is rendered as the creator of JavaScript, while the creators of React and Redux are removed. This test depicts an entire test scenario in one test case. We can move through each step – initial fetching, changing the input field value, submitting the form, and retrieving new data from the API – with the tools we've used.

React Testing Library with Jest is the most popular library combination for React testing. RTL provides relevant testing tools, while Jest has a general testing framework for test suites, test cases, assertions, and mocking capabilities. If you need an alternative to RTL, consider trying [Enzyme](https://www.robinwieruch.de/react-testing-jest-enzyme)²⁷⁴ by Airbnb.

²⁷⁴<https://www.robinwieruch.de/react-testing-jest-enzyme>

Exercises:

- Confirm your [source code](#)²⁷⁵.
 - Confirm the [changes](#)²⁷⁶.
- Read more about [React Testing Library in React](#)²⁷⁷.
- Read more about [E2E tests in React](#)²⁷⁸.
- While you continue with the upcoming sections, keep your tests green and add new tests when needed.

Snapshot Testing

Facebook created snapshot tests as a more lightweight way to test React components and their structure. A snapshot test creates an instance of your rendered component's output as HTML elements and their structure. This snapshot is compared to the same snapshot in the next test to give more output on how the rendered component changed and show why any tests failed in the difference. You can accept or deny any differences in your source code until the component functions as intended.

Snapshot tests are lightweight, with less focus on the implementation details of the component. Let's perform a snapshot test for our SearchForm component:

src/App.test.js

```
describe('SearchForm', () => {  
  
  ...  
  
  test('renders snapshot', () => {  
    const { container } = render(<SearchForm {...searchFormProps} />);  
    expect(container.firstChild).toMatchSnapshot();  
  });  
});
```

Run these tests with `npm test` and you'll see a new `src/_snapshots_` folder has been created in your project folder. Similar to RTL's `debug` function, there's a snapshot of your rendered SearchForm component as an HTML element structure in the file. Next, head to `src/App.js` file and change the HTML. For example, try removing the bold text from the SearchForm component:

²⁷⁵<https://bit.ly/3DOSIqe>

²⁷⁶<https://bit.ly/3jf3pKM>

²⁷⁷<https://www.robinwieruch.de/react-testing-library>

²⁷⁸<https://www.robinwieruch.de/react-testing-cypress>

src/App.js

```
const SearchForm = ({
  searchTerm,
  onSearchInput,
  onSearchSubmit,
}) => (
  <form onSubmit={onSearchSubmit}>
    <InputWithLabel
      id="search"
      value={searchTerm}
      isFocused
      onChange={onSearchInput}
    >
      Search:
    </InputWithLabel>

    <button type="submit" disabled={!searchTerm}>
      Submit
    </button>
  </form>
);
```

After the next test, the command line should look similar to the following:

Command Line

```
- Snapshot
+ Received

  <label
    for="search"
  >
-   <strong>
-     Search:
-   </strong>
+   Search:
  </label>
```

This is a typical case for a breaking snapshot test. When a component's HTML structure is changed unintentionally, the snapshot test informs us in the command line. To fix it, we'd go into the *src/App.js* file and edit the SearchForm component. For intentional changes, press "u" on the command line for interactive tests and a new snapshot will be created. Try it and see how the snapshot file in your *src/_snapshots_* folder changes.

Jest stores snapshots in a folder so it can validate the difference against future snapshot tests. Users can share these snapshots across teams using version control platforms like git. This is how we make sure the DOM stays the same.

Snapshot tests are useful for setting up tests quickly in React, though it's best to avoid using them exclusively. Instead, use snapshot tests for components that don't update often, are less complex, and where it's easier to compare component results.

Exercises:

- Confirm your [source code](#)²⁷⁹.
 - Confirm the [changes](#)²⁸⁰.
- Add one snapshot test for each of all the other components and check the content of your `_snapshots_` folder.
- Optional: [Rate this section](#)²⁸¹.

²⁷⁹<https://bit.ly/3G7tF3M>

²⁸⁰<https://bit.ly/3lUYpge>

²⁸¹<https://forms.gle/tMJyXvxS1AmRvSUU9>

React Project Structure

With multiple React components in one file, you might wonder why we didn't put components into different files from the start. We already have multiple components in the *src/App.js* file that can be defined in their own files/folders (sometimes also called modules). For learning, it's more practical to keep these components in one place. Once your application grows, consider splitting these components into multiple files/folders/modules so it scales properly.

Before we restructure our React project, recap [JavaScript's import and export statements](#)²⁸². Importing and exporting files are two fundamental concepts in JavaScript you must learn before React. There's no right way to structure a React application, as they evolve naturally along with the project's structure. We'll complete a simple refactoring for the project's folder/file structure for the sake of learning about the process. Afterward, there will be a few additional options about restructuring this project or React projects in general. You can continue with the restructured project, though we'll continue developing with the *src/App.js* file to keep things simple.

On the command line in your project's folder, navigate into the *src/* folder and create the following component dedicated files:

Command Line

```
touch src/List.js src/InputWithLabel.js src/SearchForm.js
```

Move every component from the *src/App.js* file in its own file, except for the List component which has to share its place with the Item component in the *src/List.js* file. Then in every file make sure to import React and to export the component which needs to be used from the file. For example, in *src/List.js* file:

src/List.js

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

const List = ({ list, onRemoveItem }) => (
  <ul>
    {list.map((item) => (
      <Item
        key={item.objectID}
        item={item}
        onRemoveItem={onRemoveItem}
      />
    ))}
  </ul>
);
```

²⁸²<https://www.robinwieruch.de/javascript-import-export>

```

const Item = ({ item, onRemoveItem }) => (
  <li>
    <span>
      <a href={item.url}>{item.title}</a>
    </span>
    <span>{item.author}</span>
    <span>{item.num_comments}</span>
    <span>{item.points}</span>
    <span>
      <button type="button" onClick={() => onRemoveItem(item)}>
        Dismiss
      </button>
    </span>
  </li>
);

export { List };

```

Since only the List component uses the Item component, we can keep it in the same file. If this changes because the Item component is used elsewhere, we can give the Item component its own file. The SearchForm component in the *src/SearchForm.js* file must import the InputWithLabel component. Like the Item component, we could have left the InputWithLabel component next to the SearchForm; but our goal is to make InputWithLabel component reusable with other components:

src/SearchForm.js

```

import * as React from 'react';

import { InputWithLabel } from './InputWithLabel';

const SearchForm = ({
  searchTerm,
  onSearchInput,
  onSearchSubmit,
}) => (
  <form onSubmit={onSearchSubmit}>
    <InputWithLabel
      id="search"
      value={searchTerm}
      isFocused
      onChange={onSearchInput}
    >
      <strong>Search:</strong>
    </InputWithLabel>
  </form>
);

```

```
    <button type="submit" disabled={!searchTerm}>
      Submit
    </button>
  </form>
);

export { SearchForm };
```

The App component has to import all the components it needs to render. It doesn't need to import InputWithLabel, because it's only used for the SearchForm component.

src/App.js

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

import { SearchForm } from './SearchForm';
import { List } from './List';

...

const App = () => {
  ...
};

export default App;
```

Components that are used in other components now have their own file. If a component should be used as a reusable component (e.g. InputWithLabel), it receives its own file. Only if a component (e.g. Item) is dedicated to another component (e.g. List) do we keep it in the same file. From here, there are several strategies to structure your folder/file hierarchy. One scenario is to create a folder for every component:

Project Structure

```
- List/  
-- index.js  
- SearchForm/  
-- index.js  
- InputWithLabel/  
-- index.js
```

The *index.js* file holds the implementation details for the component, while other files in the same folder have different responsibilities like styling, testing, and types:

Project Structure

```
- List/  
-- index.js  
-- style.css  
-- test.js  
-- types.js
```

If using CSS-in-JS, where no CSS file is needed, one could still have a separate *style.js* file for all the styled components:

Project Structure

```
- List/  
-- index.js  
-- style.js  
-- test.js  
-- types.js
```

Sometimes we'll need to move from a **technical-oriented folder structure** to a **domain-oriented folder structure**, especially once the project grows. Universal *shared/* folder is shared across domain specific components:

Project Structure

```
- Messages.js  
- Users.js  
- shared/  
-- Button.js  
-- Input.js
```

If you scale this to the deeper level folder structure, each component will have its own folder in a domain-oriented project structure as well:

Project Structure

```
- Messages/  
-- index.js  
-- style.css  
-- test.js  
-- types.js  
- Users/  
-- index.js  
-- style.css  
-- test.js  
-- types.js  
- shared/  
-- Button/  
--- index.js  
--- style.css  
--- test.js  
--- types.js  
-- Input/  
--- index.js  
--- style.css  
--- test.js  
--- types.js
```

There are many ways on how to structure your React project from small to large project: simple to complex folder structure; one-level nested to two-level nested folder nesting; dedicated folders for styling, types, and testing next to implementation logic. There is no right way for folder/file structures. However, in the exercises, you will find my 5 steps approach to structure a React project. After all, a project's requirements evolve over time and so should its structure. If keeping all assets in one file feels right, then there is no rule against it.

Exercises:

- Confirm your [source code](#)²⁸³.
 - Confirm the [changes](#)²⁸⁴.
- Read more about [JavaScript's import and export statements](#)²⁸⁵.
- Read more about [React Folder Structures](#)²⁸⁶.
- Keep the current folder structure if you feel confident. The ongoing sections will omit it, only using the `src/App.js` file.
- Optional: [Rate this section](#)²⁸⁷.

²⁸³<https://bit.ly/2XxzDcG>

²⁸⁴<https://bit.ly/3DW8109>

²⁸⁵<https://www.robinwieruch.de/javascript-import-export>

²⁸⁶<https://www.robinwieruch.de/react-folder-structure>

²⁸⁷<https://forms.gle/YLzszsmtdB1DQBCe7>

Real World React (Advanced)

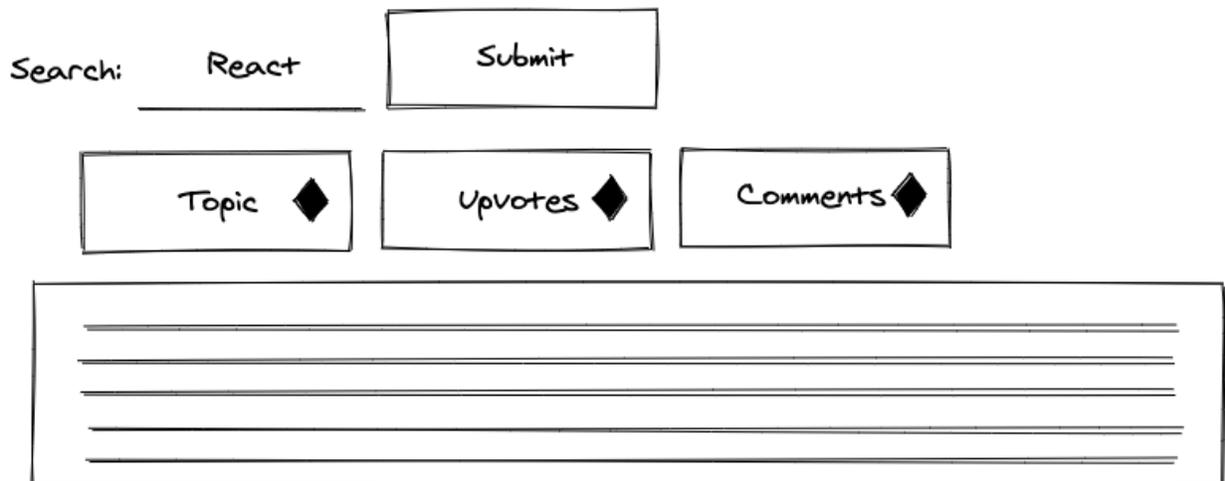
We've covered most of React's fundamentals, its legacy features, and techniques for maintaining applications. Now it's time to dive into developing real-world React features. Each of the following sections will come with a task. Try to tackle these tasks without the *optional hints* first, but be aware that these are going to be challenging on your first attempt. If you need help, use the *optional hints* or follow the instructions from the section.

Sorting

Task: Working with a list of items often includes interactions that make data more approachable by users. So far, every item was listed with each of its properties. To make it explorable, the list should enable the sorting of each property by title, author, comments, and points in ascending or descending order. Sorting in only one direction is fine, because sorting in the other direction will be part of the next task.

Optional Hints:

- Introduce a new sort state in the App or List component.
- For each property (e.g. title, author, points, num_comments) implement an HTML button which sets the sort state for this property.
- Use the sort state to apply an appropriate sort function on the list.
- Using a utility library like [Lodash](https://lodash.com)²⁸⁸ for its `sortBy` function is encouraged.



Okay, let's tackle this task! We will treat the list of data like a table. Each row represents an item of the list and each column represents one property of the item. Introducing headers should provide the user more guidance about each column:

²⁸⁸<https://lodash.com>

src/App.js

```

const List = ({ list, onRemoveItem }) => (
  <ul>
    <li style={{ display: 'flex' }}>
      <span style={{ width: '40%' }}>Title</span>
      <span style={{ width: '30%' }}>Author</span>
      <span style={{ width: '10%' }}>Comments</span>
      <span style={{ width: '10%' }}>Points</span>
      <span style={{ width: '10%' }}>Actions</span>
    </li>

    {list.map((item) => (
      <Item
        key={item.objectID}
        item={item}
        onRemoveItem={onRemoveItem}
      />
    ))}
  </ul>
);

```

We are using inline style for the most basic layout. To match the layout of the header with the rows, give the rows in the Item component a layout as well:

src/App.js

```

const Item = ({ item, onRemoveItem }) => (
  <li style={{ display: 'flex' }}>
    <span style={{ width: '40%' }}>
      <a href={item.url}>{item.title}</a>
    </span>
    <span style={{ width: '30%' }}>{item.author}</span>
    <span style={{ width: '10%' }}>{item.num_comments}</span>
    <span style={{ width: '10%' }}>{item.points}</span>
    <span style={{ width: '10%' }}>
      <button type="button" onClick={() => onRemoveItem(item)}>
        Dismiss
      </button>
    </span>
  </li>
);

```

In the ongoing implementation, we will remove the style attributes, because it takes up lots of space and clutters the actual implementation logic (hence extracting it into proper CSS). But I encourage you to keep it for yourself.

The List component will handle the new sort state. This can also be done in the App component, but in the end, only the List component needs it, so we can lift the state directly to it. The sort state initializes with a 'NONE' state, so the list items are displayed in the order they are fetched from the API. Furthermore, we will add a new handler to set the sort state with a sort-specific key:

src/App.js

```
const List = ({ list, onRemoveItem }) => {
  const [sort, setSort] = React.useState('NONE');

  const handleSort = (sortKey) => {
    setSort(sortKey);
  };

  return (
    ...
  );
};
```

In the List component's header, buttons can help us to set the sort state for each column/property. An inline handler is used to sneak in the sort-specific key (`sortKey`). When the button for the "Title" column is clicked, 'TITLE' becomes the new sort state:

src/App.js

```
const List = ({ list, onRemoveItem }) => {
  ...

  return (
    <div>
      <div>
        <span>
          <button type="button" onClick={() => handleSort('TITLE')}>
            Title
          </button>
        </span>
        <span>
          <button type="button" onClick={() => handleSort('AUTHOR')}>
            Author
          </button>
        </span>
      </div>
    </div>
  );
};
```

```

    <span>
      <button type="button" onClick={() => handleSort('COMMENT')}>
        Comments
      </button>
    </span>
    <span>
      <button type="button" onClick={() => handleSort('POINT')}>
        Points
      </button>
    </span>
    <span>Actions</span>
  </div>

  {list.map((item) => ... )}
</div>
);
};

```

The state management for the new feature is implemented, but we don't see anything when our buttons are clicked yet. This happens because the sorting mechanism hasn't been applied to the actual `list`. Sorting an array with JavaScript isn't trivial, because every JavaScript primitive (e.g. string, boolean, number) comes with edge cases when an array is sorted by its properties. We will use a library called [Lodash](#)²⁸⁹ to solve this, which comes with many JavaScript utility functions (e.g. `sortBy`). First, install it via the command line:

Command Line

```
npm install lodash
```

Second, at the top of your file, import the utility function for sorting:

src/App.js

```
import * as React from 'react';
import axios from 'axios';
import { sortBy } from 'lodash';
```

```
...
```

Third, create a JavaScript object (also called dictionary in this case) with all the possible `sortKey` and sort function mappings. Each specific sort key is mapped to a function that sorts the incoming `list`. Sorting by `'NONE'` returns the unsorted list; sorting by `'POINT'` returns a list and its items sorted by the `points` property, and so on:

²⁸⁹<https://lodash.com>

src/App.js

```
const SORTS = {
  NONE: (list) => list,
  TITLE: (list) => sortBy(list, 'title'),
  AUTHOR: (list) => sortBy(list, 'author'),
  COMMENT: (list) => sortBy(list, 'num_comments').reverse(),
  POINT: (list) => sortBy(list, 'points').reverse(),
};

const List = ({ list, onRemoveItem }) => {
  ...
};
```

With the sort (sortKey) state and all possible sort variations (SORTS) at our disposal, we can sort the list before mapping it:

src/App.js

```
const List = ({ list, onRemoveItem }) => {
  const [sort, setSort] = React.useState('NONE');

  const handleSort = (sortKey) => {
    setSort(sortKey);
  };

  const sortFunction = SORTS[sort];
  const sortedList = sortFunction(list);

  return (
    <div>
      ...

      {sortedList.map((item) => (
        <Item
          key={item.objectID}
          item={item}
          onRemoveItem={onRemoveItem}
        />
      ))}
    </div>
  );
};
```

Task's done and here comes the recap: First we extracted the sort function from the dictionary by its `sortKey` (state), then we applied the function to the list before mapping it to render each `Item` component. Second, we rendered HTML buttons as header columns to give our users interaction. Then, we added implementation details for each button by changing the sort state. Finally, we used the sort state to sort the actual list.

Exercises:

- Confirm your [source code](#)²⁹⁰.
 - Confirm the [changes](#)²⁹¹.
- Read more about [Lodash](#)²⁹².
- Why did we use numeric properties like `points` and `num_comments` for a reverse sort?
- Use your styling skills to give the user feedback about the current active sort. This mechanism can be as straightforward as giving the active sort button a different color.
- Optional: [Rate this section](#)²⁹³.

²⁹⁰<https://bit.ly/3aUfX5W>

²⁹¹<https://bit.ly/2ZfbVTb>

²⁹²<https://lodash.com>

²⁹³<https://forms.gle/GM71SDZZWPQWEwmB7>

Reverse Sort

Task: The sort feature works, but the ordering only includes one direction. Implement a reverse sort when a sort button is clicked twice, so it becomes a toggle between normal (ascending) and reverse (descending) sort.

Optional Hints:

- Consider that reverse or normal sort could be just another state (e.g. `isReverse`) next to the `sortKey`.
- Set the new state in the `handleSort` handler based on the previous sort.
- Use the new `isReverse` state for sorting the list with the sort function from the dictionary with the optionally applied `reverse()` function from JavaScript arrays.

Let's get to the task. The initial sort direction works for strings, as well as numeric sorts like the reverse sort for JavaScript numbers that arranges them from high to low. Now we need another state to track whether the sort is reversed or normal:

src/App.js

```
const List = ({ list, onRemoveItem }) => {
  const [sort, setSort] = React.useState({
    sortKey: 'NONE',
    isReverse: false,
  });

  ...
};
```

Next, give the sort handler logic to see if the incoming `sortKey` triggers are a normal or reverse sort. If the `sortKey` is the same as the one in the state, it should be a reverse sort, but only if the sort state wasn't already reversed:

src/App.js

```
const List = ({ list, onRemoveItem }) => {
  const [sort, setSort] = React.useState({
    sortKey: 'NONE',
    isReverse: false,
  });

  const handleSort = (sortKey) => {
    const isReverse = sort.sortKey === sortKey && !sort.isReverse;
```

```
    setSort({ sortKey: sortKey, isReverse: isReverse });
  };

  const sortFunction = SORTS[sort.sortKey];
  const sortedList = sortFunction(list);

  return (
    ...
  );
};
```

Lastly, depending on the new `isReverse` state, apply the sort function from the dictionary with or without the built-in JavaScript reverse method for arrays:

`src/App.js`

```
const List = ({ list, onRemoveItem }) => {
  const [sort, setSort] = React.useState({
    sortKey: 'NONE',
    isReverse: false,
  });

  const handleSort = (sortKey) => {
    const isReverse = sort.sortKey === sortKey && !sort.isReverse;

    setSort({ sortKey, isReverse });
  };

  const sortFunction = SORTS[sort.sortKey];
  const sortedList = sort.isReverse
    ? sortFunction(list).reverse()
    : sortFunction(list);

  return (
    ...
  );
};
```

The reverse sort is now operational! Congratulations, you have a fully sortable list now. And by the way: For the object passed to the state updater function, we use what is called a **shorthand object initializer notation**:

src/App.js

```
const firstName = 'Robin';

const user = {
  firstName: firstName,
};

console.log(user);
// { firstName: "Robin" }
```

When the property name in your object is the same as your variable name, you can omit the key/value pair and just write the name:

src/App.js

```
const firstName = 'Robin';

const user = {
  firstName,
};

console.log(user);
// { firstName: "Robin" }
```

If necessary, read more about [JavaScript Object Initializers](#)²⁹⁴.

Exercises:

- Confirm your [source code](#)²⁹⁵.
 - Confirm the [changes](#)²⁹⁶.
- Consider the drawback of keeping the sort state in the List instead of the App component. If you don't know, sort the list by "Title" and search for other stories afterward. What would be different if the sort state would be in the App component.
- Use your styling skills to give the user feedback about the current active sort and its reverse state. It could be an [arrow up or arrow down SVG](#)²⁹⁷ next to each active sort button.
- Optional: [Rate this section](#)²⁹⁸.

²⁹⁴<https://mzl.la/2XuN651>

²⁹⁵<https://bit.ly/3lVe7Ib>

²⁹⁶<https://bit.ly/3jerTE3>

²⁹⁷<https://bit.ly/3lXfLZN>

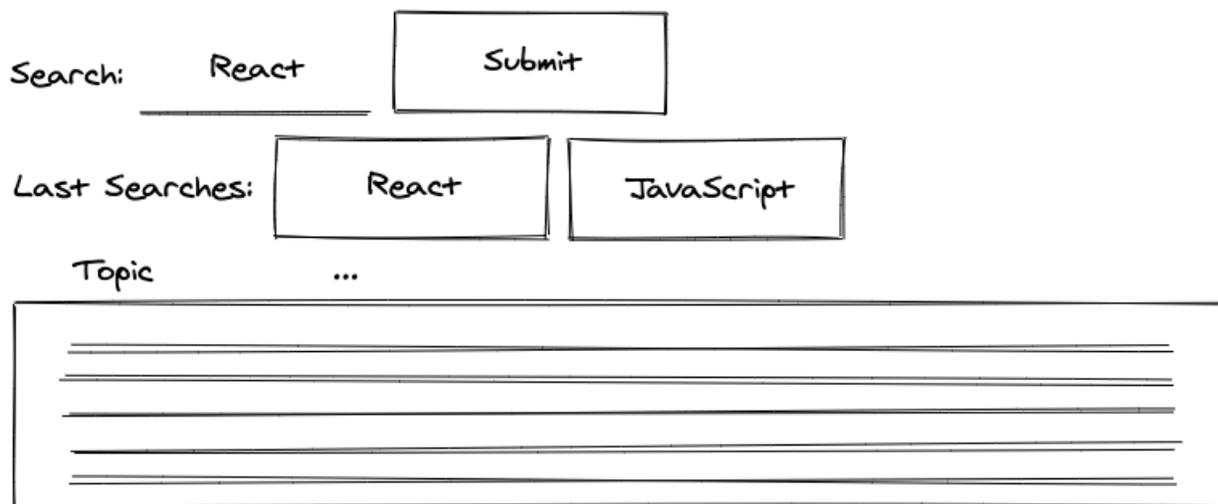
²⁹⁸<https://forms.gle/ZoJSHFJf2swcBHXM6>

Remember Last Searches

Task: Remember the last five search terms which hit the API, and provide a button to move quickly between searches. When the buttons are clicked, stories for the search term should be fetched again.

Optional Hints:

- Don't use a new state for this feature. Instead, reuse the `url` state and `setUrl` state updater function to fetch stories from the API. Adapt them to multiple `urls` as state, and to set multiple `urls` with `setUrls`. The last URL from `urls` can be used to fetch the data, and the last five URLs from `urls` can be used to display the buttons.



Let's get to it. First, we will refactor all `url` to `urls` state and all `setUrl` to `setUrls` state updater functions. Instead of initializing the state with an `url` as a string, make it an array with the initial `url` as its only entry:

`src/App.js`

```
const App = () => {
  ...

  const [urls, setUrls] = React.useState([
    `${API_ENDPOINT}${searchTerm}`,
  ]);

  ...
};
```

Second, instead of using the current `url` state for data fetching, use the last `url` entry from the `urls` array. If another `url` is added to the list of `urls`, it is used to fetch data instead:

src/App.js

```
const App = () => {  
  
  ...  
  
  const handleFetchStories = React.useCallback(async () => {  
    dispatchStories({ type: 'STORIES_FETCH_INIT' });  
  
    try {  
      const lastUrl = urls[urls.length - 1];  
      const result = await axios.get(lastUrl);  
  
      dispatchStories({  
        type: 'STORIES_FETCH_SUCCESS',  
        payload: result.data.hits,  
      });  
    } catch {  
      dispatchStories({ type: 'STORIES_FETCH_FAILURE' });  
    }  
  }, [urls]);  
  
  ...  
};
```

And third, instead of storing the `url` string as state with the state updater function, concatenate the new `url` using the `concat` method with the previous `urls` in an array for the new state:

src/App.js

```
const App = () => {  
  
  ...  
  
  const handleSearchSubmit = (event) => {  
    const url = `${API_ENDPOINT}${searchTerm}`;  
    setUrls(urls.concat(url));  
  
    event.preventDefault();  
  };  
  
  ...  
};
```

With each search, another URL is stored in our state of `urls`. Next, render a button for each of the last five URLs. We'll include a new universal handler for these buttons, and each passes a specific `url` with a more specific inline handler:

`src/App.js`

```
const getLastSearches = (urls) => urls.slice(-5);
```

```
...
```

```
const App = () => {
```

```
  ...
```

```
  const handleLastSearch = (url) => {  
    // do something  
  };
```

```
  const lastSearches = getLastSearches(urls);
```

```
  return (  
    <div>
```

```
      <h1>My Hacker Stories</h1>
```

```
      <SearchForm ... />
```

```
      {lastSearches.map((url) => (  
        <button  
          key={url}  
          type="button"  
          onClick={() => handleLastSearch(url)}  
        >  
          {url}  
        </button>  
      )}}
```

```
      ...
```

```
    </div>
```

```
  );
```

```
};
```

Next, instead of showing the whole URL of the last search in the button as button text, show only the search term by replacing the API's endpoint with an empty string:

src/App.js

```
const extractSearchTerm = (url) => url.replace(API_ENDPOINT, '');

const getLastSearches = (urls) =>
  urls.slice(-5).map((url) => extractSearchTerm(url));

...

const App = () => {
  ...

  const lastSearches = getLastSearches(urls);

  return (
    <div>
      ...

      {lastSearches.map((searchTerm) => (
        <button
          key={searchTerm}
          type="button"
          onClick={() => handleLastSearch(searchTerm)}
        >
          {searchTerm}
        </button>
      ))}

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

The `getLastSearches` function now returns search terms instead of URLs. The actual `searchTerm` is passed to the inline handler instead of the `url`. By mapping over the list of `urls` in `getLastSearches`, we can extract the search term for each `url` within the array's `map` method. Making it more concise, it can also look like this:

src/App.js

```
const getLastSearches = (urls) =>
  urls.slice(-5).map(extractSearchTerm);
```

Now we'll provide functionality for the new handler used by every button, since clicking one of these buttons should trigger another search. Since we use the `urls` state for fetching data, and since we know the last URL is always used for data fetching, concatenate a new `url` to the list of `urls` to trigger another search request:

src/App.js

```
const App = () => {
  ...

  const handleLastSearch = (searchTerm) => {
    const url = `${API_ENDPOINT}${searchTerm}`;
    setUrls(urls.concat(url));
  };

  ...
};
```

If you compare this new handler's implementation logic to the `handleSearchSubmit`, you may see some common functionality. Extract this common functionality to a new handler and a new extracted utility function:

src/App.js

```
const getUrl = (searchTerm) => `${API_ENDPOINT}${searchTerm}`;

...

const App = () => {
  ...

  const handleSearchSubmit = (event) => {
    handleSearch(searchTerm);

    event.preventDefault();
  };

  const handleLastSearch = (searchTerm) => {
    handleSearch(searchTerm);
```

```
};

const handleSearch = (searchTerm) => {
  const url = getUrl(searchTerm);
  setUrls(urls.concat(url));
};

...
};
```

The new utility function can be used somewhere else in the App component. If you extract functionality that can be used by two parties, always check to see if it can be used by a third party:

src/App.js

```
const App = () => {
  ...

  // important: still wraps the returned value in []
  const [urls, setUrls] = React.useState([getUrl(searchTerm)]);

  ...
};
```

The functionality should work, but it complains or breaks if the same search term is used more than once, because `searchTerm` is used for each button element as key attribute. Make the key more specific by concatenating it with the `index` of the mapped array.

src/App.js

```
const App = () => {
  ...

  return (
    <div>
      ...

      {lastSearches.map((searchTerm, index) => (
        <button
          key={searchTerm + index}
          type="button"
          onClick={() => handleLastSearch(searchTerm)}
        >
          {searchTerm}
        </button>
      )}
    </div>
  );
};
```

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

It's not the perfect solution, because the `index` isn't a stable key (especially when adding items to the list); however, it doesn't break in this scenario. The feature works now, but you can add further UX improvements by following the tasks below.

More Tasks:

- (1) Do not show the current search as a button, only the five preceding searches. Hint: Adapt the `getLastSearches` function.
- (2) Don't show duplicated searches. Searching twice for "React" shouldn't create two different buttons. Hint: Adapt the `getLastSearches` function.
- (3) Set the `SearchForm` component's input field value with the last search term if one of the buttons is clicked.

The source of the five rendered buttons is the `getLastSearches` function. There, we take the array of `urls` and return the last five entries from it. Now we'll change this utility function to return the last six entries instead of five by removing the last one, in order to not show the current search as a button. Afterward, only the five *previous* searches are displayed as buttons:

`src/App.js`

```
const getLastSearches = (urls) =>
  urls
    .slice(-6)
    .slice(0, -1)
    .map(extractSearchTerm);
```

If the same search is executed two or more times in a row, duplicate buttons appear, which is likely not your desired behavior. It would be acceptable to group identical searches into one button if they followed each other. We will solve this problem in the utility function as well. Before separating the array into the five previous searches, group the identical searches:

src/App.js

```
const getLastSearches = (urls) =>
  urls
    .reduce((result, url, index) => {
      const searchTerm = extractSearchTerm(url);

      if (index === 0) {
        return result.concat(searchTerm);
      }

      const previousSearchTerm = result[result.length - 1];

      if (searchTerm === previousSearchTerm) {
        return result;
      } else {
        return result.concat(searchTerm);
      }
    }, [])
    .slice(-6)
    .slice(0, -1);
```

The reduce function starts with an empty array as its `result`. The first iteration concatenates the `searchTerm` we extracted from the first `url` into the `result`. Every extracted `searchTerm` is compared to the one before it. If the previous search term is different from the current, concatenate the `searchTerm` to the result. If the search terms are identical, return the result without adding anything.

The `SearchForm` component's input field should be set with the new `searchTerm` if one of the last search buttons is clicked. We can solve this using the state updater function for the specific value used in the `SearchForm` component.

src/App.js

```
const App = () => {
  ...

  const handleLastSearch = (searchTerm) => {
    setSearchTerm(searchTerm);

    handleSearch(searchTerm);
  };

  ...
};
```

Lastly, extract the feature's new rendered content from this section as a standalone component, to keep the App component lightweight:

src/App.js

```
const App = () => {
  ...

  const lastSearches = getLastSearches(urls);

  return (
    <div>
      ...

      <SearchForm ... />

      <LastSearches
        lastSearches={lastSearches}
        onLastSearch={handleLastSearch}
      />

      <hr />

      ...
    </div>
  );
};

const LastSearches = ({ lastSearches, onLastSearch }) => (
  <>
    {lastSearches.map((searchTerm, index) => (
      <button
        key={searchTerm + index}
        type="button"
        onClick={() => onLastSearch(searchTerm)}
      >
        {searchTerm}
      </button>
    ))}
  </>
);
```

This feature wasn't an easy one. Lots of fundamental React but also JavaScript knowledge was needed to accomplish it. If you had no problems implementing it yourself or in following the

instructions, you are very well set. If you had one or the other issue, don't worry too much about it. Maybe you even figured out another way to solve this task and it may have turned out simpler than the one I showed here.

Exercises:

- Confirm your [source code](#)²⁹⁹.
 - Confirm the [changes](#)³⁰⁰.
- Read more about [grouping in JavaScript](#)³⁰¹.
- Optional: [Rate this section](#)³⁰².

²⁹⁹<https://bit.ly/3vtAYgZ>

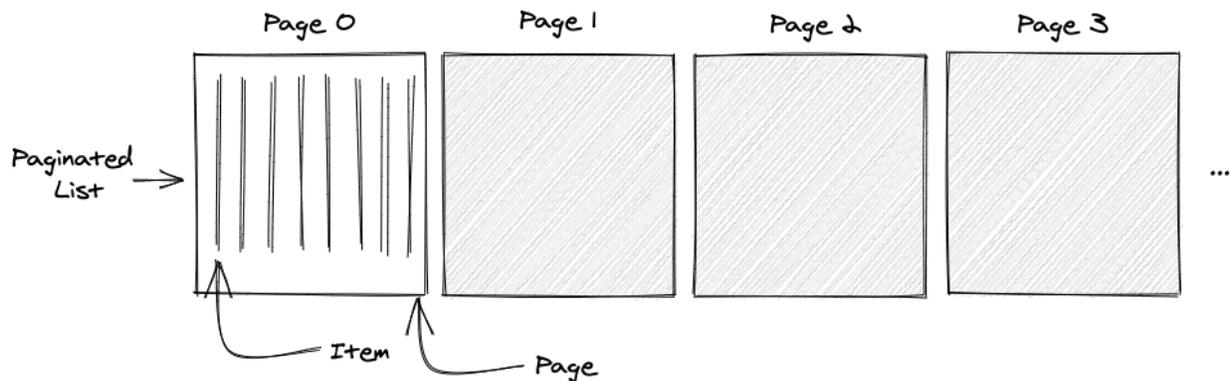
³⁰⁰<https://bit.ly/3jlnHm1>

³⁰¹<https://www.robinwieruch.de/javascript-groupby/>

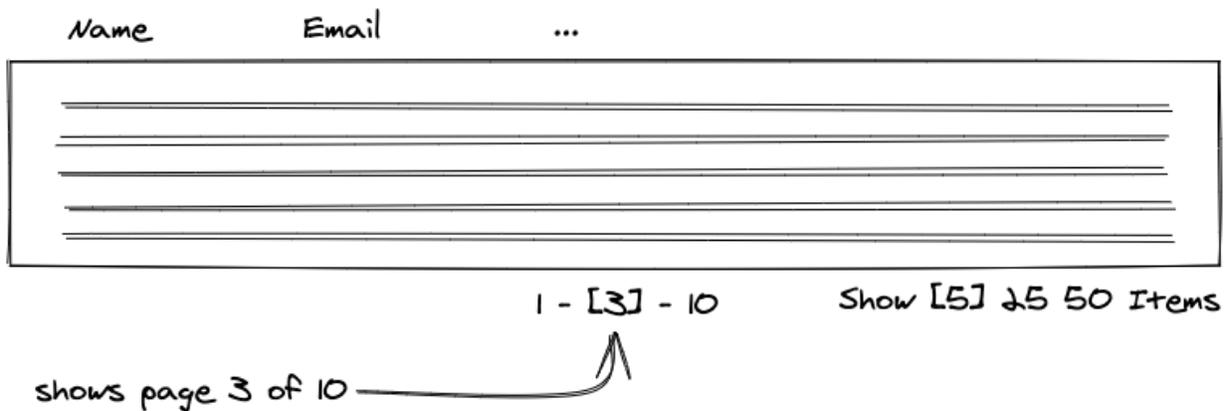
³⁰²<https://forms.gle/LhNVodZgu8qTqHhN6>

Paginated Fetch

Searching for popular stories via Hacker News API is only one step towards a fully functional search engine, and there are many ways to fine-tune the search. Take a closer look at the data structure and observe how [the Hacker News API³⁰³](https://hn.algolia.com/api) returns more than a list of hits. Specifically, it returns a paginated list. The page property, which is 0 in the first response, can be used to fetch more paginated lists as results. You only need to pass the next page with the same search term to the API.

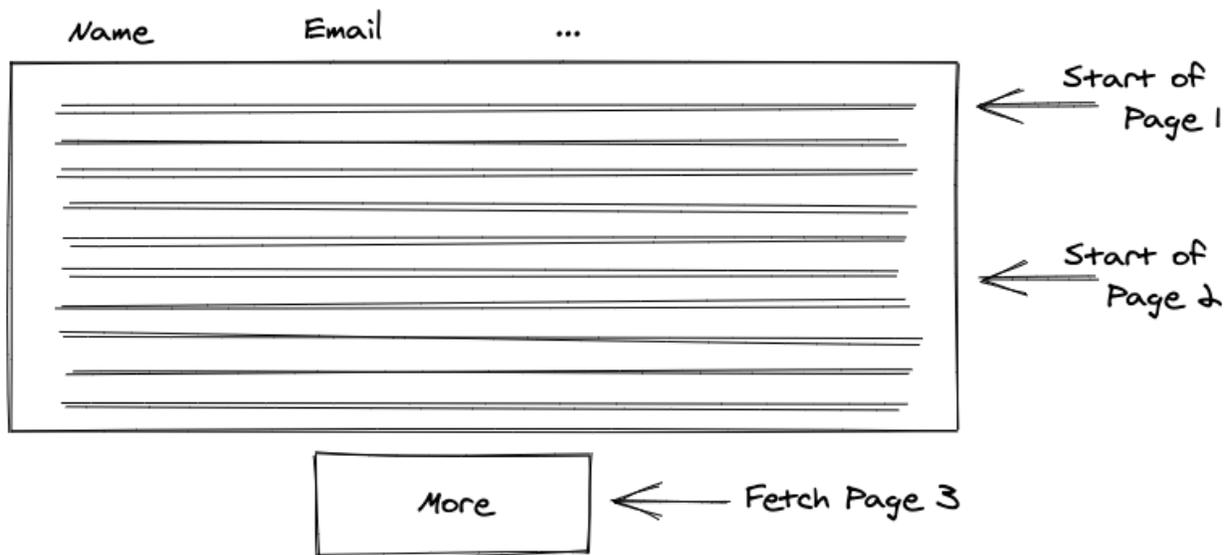


The following shows how to implement a paginated fetch with the Hacker News data structure. If you are used to **pagination** from other applications, you may have a row of buttons from 1-10 in your mind – where the currently selected page is highlighted 1-[3]-10 and where clicking one of the buttons leads to fetching and displaying this subset of data.



In contrast, we will implement the feature as **infinite pagination**. Instead of rendering a single paginated list on a button click, we will render *all paginated lists as one list* with *one* button to fetch the next page. Every additional *paginated list* is concatenated at the end of the *one list*.

³⁰³<https://hn.algolia.com/api>



Task: Rather than fetching only the first page of a list, extend the functionality for fetching succeeding pages. Implement this as infinite pagination on button click.

Optional Hints:

- Extend the `API_ENDPOINT` with the parameters needed for the paginated fetch.
- Store the `page` from the `result` as state after fetching the data.
- Fetch the first page (0) of data with every search.
- Fetch the succeeding page (`page + 1`) for every additional request triggered with a new HTML button.

Let's do this! First, extend the API constant so it can deal with paginated data later. We will turn this one constant:

`src/App.js`

```
const API_ENDPOINT = 'https://hn.algolia.com/api/v1/search?query=';
```

```
const getUrl = (searchTerm) => `${API_ENDPOINT}${searchTerm}`;
```

Into a composable API constant with its parameters:

src/App.js

```
const API_BASE = 'https://hn.algolia.com/api/v1';
const API_SEARCH = '/search';
const PARAM_SEARCH = 'query=';

// careful: notice the ? in between
const getUrl = (searchTerm) =>
  `${API_BASE}${API_SEARCH}?${PARAM_SEARCH}${searchTerm}`;
```

Fortunately, we don't need to adjust the API endpoints at other places of the application, because we extracted a common `getUrl` function for it. However, there is one spot where we must address this logic for the future:

src/App.js

```
const extractSearchTerm = url => url.replace(API_ENDPOINT, '');
```

In the next steps, it won't be sufficient to replace the base of our API endpoint, which is no longer in our code. With more parameters for the API endpoint, the URL becomes more complex. It will change from X to Y:

src/App.js

```
// X
https://hn.algolia.com/api/v1/search?query=react

// Y
https://hn.algolia.com/api/v1/search?query=react&page=0
```

It's better to extract the search term by extracting everything between `?` and `&`. Also consider that the `query` parameter is directly after the `?` and all other parameters like `page` follow it:

src/App.js

```
const extractSearchTerm = (url) =>
  url.substring(url.lastIndexOf('?') + 1, url.lastIndexOf('&'));
```

The key (`query=`) also needs to be replaced, leaving only the value (`searchTerm`):

src/App.js

```
const extractSearchTerm = (url) =>
  url
    .substring(url.lastIndexOf('?') + 1, url.lastIndexOf('&'))
    .replace(PARAM_SEARCH, '');
```

Essentially, we'll trim the string until we leave only the search term:

src/App.js

```
// url
https://hn.algolia.com/api/v1/search?query=react&page=0

// url after substring
query=react

// url after replace
react
```

Next, the returned result from the Hacker News API delivers us the page data:

src/App.js

```
const App = () => {
  ...

  const handleFetchStories = React.useCallback(async () => {
    dispatchStories({ type: 'STORIES_FETCH_INIT' });

    try {
      const lastUrl = urls[urls.length - 1];
      const result = await axios.get(lastUrl);

      dispatchStories({
        type: 'STORIES_FETCH_SUCCESS',
        payload: {
          list: result.data.hits,
          page: result.data.page,
        },
      });
    } catch {
      dispatchStories({ type: 'STORIES_FETCH_FAILURE' });
    }
  }, [urls]);
```

```
...  
};
```

We need to store this data to make paginated fetches later:

src/App.js

```
const storiesReducer = (state, action) => {  
  switch (action.type) {  
    case 'STORIES_FETCH_INIT':  
      ...  
    case 'STORIES_FETCH_SUCCESS':  
      return {  
        ...state,  
        isLoading: false,  
        isError: false,  
        data: action.payload.list,  
        page: action.payload.page,  
      };  
    case 'STORIES_FETCH_FAILURE':  
      ...  
    case 'REMOVE_STORY':  
      ...  
    default:  
      throw new Error();  
  }  
};  
  
const App = () => {  
  ...  
  
  const [stories, dispatchStories] = React.useReducer(  
    storiesReducer,  
    { data: [], page: 0, isLoading: false, isError: false }  
  );  
  
  ...  
};
```

Extend the API endpoint with the new page parameter. This change was supported by our premature optimizations earlier, when we extracted the search term from the URL:

src/App.js

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

// careful: notice the ? and & in between
const getUrl = (searchTerm, page) =>
  `${API_BASE}${API_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}${page}`;
```

Next, we must adjust all `getUrl` invocations by passing the `page` argument. Since the initial search and the last search always fetch the first page (0), we pass this page as an argument to the function for retrieving the appropriate URL:

src/App.js

```
const App = () => {
  ...

  const [urls, setUrls] = React.useState([getUrl(searchTerm, 0)]);

  ...

  const handleSearchSubmit = (event) => {
    handleSearch(searchTerm, 0);

    event.preventDefault();
  };

  const handleLastSearch = (searchTerm) => {
    setSearchTerm(searchTerm);

    handleSearch(searchTerm, 0);
  };

  const handleSearch = (searchTerm, page) => {
    const url = getUrl(searchTerm, page);
    setUrls(urls.concat(url));
  };

  ...
};
```

To fetch the next page when a button is clicked, we'll need to increment the `page` argument in this new handler:

`src/App.js`

```
const App = () => {
  ...

  const handleMore = () => {
    const lastUrl = urls[urls.length - 1];
    const searchTerm = extractSearchTerm(lastUrl);
    handleSearch(searchTerm, stories.page + 1);
  };

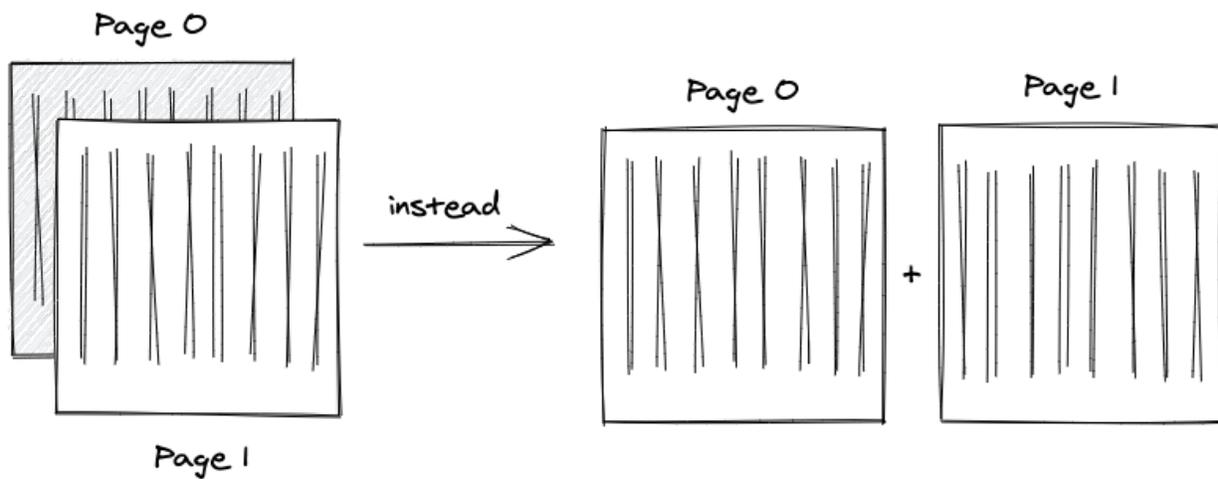
  ...

  return (
    <div>
      ...

      {stories.isLoading ? (
        <p>Loading ...</p>
      ) : (
        <List list={stories.data} onRemoveItem={handleRemoveStory} />
      )}

      <button type="button" onClick={handleMore}>
        More
      </button>
    </div>
  );
};
```

We've implemented data fetching with the dynamic `page` argument. The initial and last searches always use the first page, and every fetch with the new "More" button uses an incremented page. There is one crucial bug when trying the feature, though: the new fetches don't extend the previous list, but completely replace it.



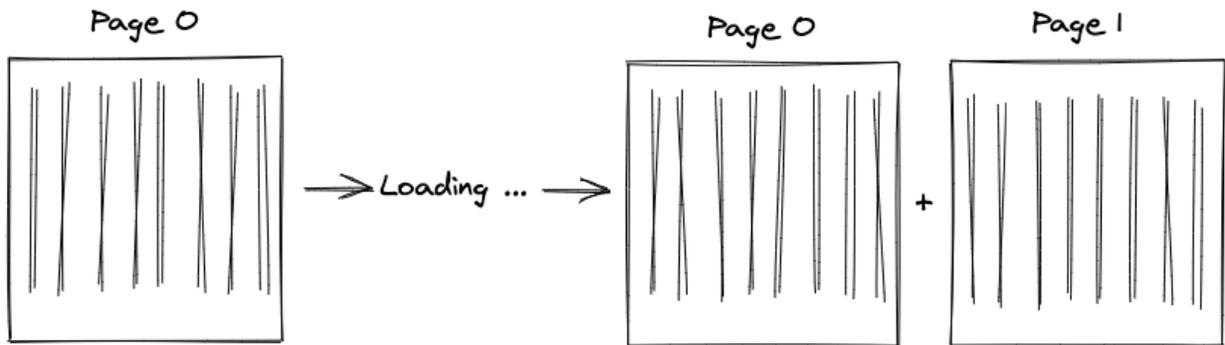
We solve this in the reducer by avoiding the replacement of current data with new data, concatenating the paginated lists:

src/App.js

```
const storiesReducer = (state, action) => {
  switch (action.type) {
    case 'STORIES_FETCH_INIT':
      ...
    case 'STORIES_FETCH_SUCCESS':
      return {
        ...state,
        isLoading: false,
        isError: false,
        data:
          action.payload.page === 0
            ? action.payload.list
            : state.data.concat(action.payload.list),
        page: action.payload.page,
      };
    case 'STORIES_FETCH_FAILURE':
      ...
    case 'REMOVE_STORY':
      ...
    default:
      throw new Error();
  }
};
```

The displayed list grows after fetching more data with the new button. However, there is still a

flicker straining the UX. When fetching paginated data, the list disappears for a moment because the loading indicator appears and reappears after the request resolves.



The desired behavior is to render the list – which is an empty list in the beginning – and replace the “More” button with the loading indicator only for the next requests. This is a common UI refactoring for conditional rendering when the task evolves from a single list to paginated lists:

src/App.js

```
const App = () => {
  ...

  return (
    <div>
      ...

      <List list={stories.data} onRemoveItem={handleRemoveStory} />

      {stories.isLoading ? (
        <p>Loading ...</p>
      ) : (
        <button type="button" onClick={handleMore}>
          More
        </button>
      )}
    </div>
  );
};
```

It’s possible to fetch ongoing data for popular stories now. When working with third-party APIs, it’s always a good idea to explore its API surface. Every remote API returns different data structures, so its features may vary.

Exercises:

- Confirm your [source code](#)³⁰⁴.
 - Confirm the [changes](#)³⁰⁵.
- Revisit the [Hacker News API documentation](#)³⁰⁶: Is there a way to fetch more items in a list for a page by just adding further parameters to the API endpoint?
- Revisit the beginning of this section which speaks about pagination and infinite pagination. How would you implement a normal pagination component with buttons from 1-[3]-10, where each button fetches and displays only one page of the list?
- Instead of having one “More” button, how would you implement infinite pagination with an infinite scroll technique? Rather than clicking a button for fetching the next page explicitly, the infinite scroll could fetch the next page once the viewport of the browser hits the bottom of the displayed list.
- Optional: [Rate this section](#)³⁰⁷.

³⁰⁴<https://bit.ly/3plSBym>

³⁰⁵<https://bit.ly/3vu8OT1>

³⁰⁶<https://hn.algolia.com/api>

³⁰⁷<https://forms.gle/maPsfzHLba8gheCQA>

Deploying a React Application

Now it's time to get out into the world with your React application. There are many ways to deploy a React application to production, and many competing providers that offer this service. We'll keep it simple here by narrowing it down on one provider, after which you'll be equipped to check out other hosting providers on your own.

Build Process

So far, everything we've done has been the *development stage* of the application, when the development server handles everything: packaging all files to one application and serving it on localhost on your local machine. As a result, our code isn't available for anyone else.

The next step is to take your application to the *production stage* by hosting it on a remote server, called deployment, making it accessible for users of your application. Before an application can go public, it needs to be packaged as one essential application. Redundant code, testing code, and duplications are removed. There is also a process called minification at work which reduces the code size once more.

Fortunately, optimizations and packaging, also called bundling, comes with the build tools in create-react-app. First, build your application on the command line:

Command Line

```
npm run build
```

This creates a new *build/* folder in your project with the bundled application. You could take this folder and deploy it on a hosting provider now, but we'll use a local server to mimic this process before engaging in the real thing. First, install an HTTP server on your machine:

Command Line

```
npm install -g http-server
```

Next, serve your application with this local HTTP server:

Command Line

```
http-server build/
```

The process can also be done on demand with a single command:

Command Line

```
npx http-server build/
```

After entering one of the commands, a URL is presented that provides access to your optimized, packaged and hosted application. It's sent through a local IP address that can be made available over your local network, meaning we're hosting the application on our local machine.

Exercises:

- Optional: [Rate this section](#)³⁰⁸.

³⁰⁸<https://forms.gle/hFsut8q7eYsWfYL7A>

Deploy to Firebase

After we've built a full-fledged application in React, the final step is deployment. It is the tipping point of getting your ideas into the world, from learning how to code to producing applications. We will use Firebase Hosting for deployment.

Firebase works for create-react-app, as well as most libraries and frameworks like Angular and Vue. First, install the Firebase CLI globally to the node modules:

Command Line

```
npm install -g firebase-tools
```

Using a global installation of the Firebase CLI lets us deploy applications without concern over project dependency. For any globally-installed node package, remember to update it to a newer version with the same command as often as you can:

Command Line

```
npm install -g firebase-tools
```

If you don't have a Firebase project yet, sign up for a [Firebase account](#)³⁰⁹ and create a new project there. Then you can associate the Firebase CLI with the Firebase account (Google account):

Command Line

```
firebase login
```

A URL will display in the command line that you can open in a browser, or the Firebase CLI opens it. Choose a Google account to create a Firebase project, and give Google the necessary permissions. Return to the command line to verify a successful login.

Next, move to the project's folder and execute the following command, which initializes a Firebase project for the Firebase hosting features:

Command Line

```
firebase init
```

Next, choose the Hosting option. If you're interested in using another tool next to Firebase Hosting, add other options:

³⁰⁹<https://console.firebase.google.com>

Command Line

```
? Which Firebase CLI features do you want to set up for this folder? Press Space to \
select features, then Enter to confirm your choices.
? Database: Deploy Firebase Realtime Database Rules
? Firestore: Deploy rules and create indexes for Firestore
? Functions: Configure and deploy Cloud Functions
-> Hosting: Configure and deploy Firebase Hosting sites
? Storage: Deploy Cloud Storage security rules
```

Google becomes aware of all Firebase projects associated with an account after login, and we can select one from the Firebase platform:

Command Line

```
First, let's associate this project directory with a Firebase project.
You can create multiple project aliases by running firebase use --add,
but for now we'll just set up a default project.
```

```
? Select a default Firebase project for this directory:
-> my-react-project-abc123 (my-react-project)
i Using project my-react-project-abc123 (my-react-project)
```

There are a few other configuration steps to define. Instead of using the default *public/* folder, we want to use the *build/* folder from create-react-app. If you set up the bundling with a tool like Webpack yourself, you can choose the appropriate name for the build folder:

Command Line

```
? What do you want to use as your public directory? build
? Configure as a single-page app (rewrite all urls to /index.html)? Yes
? File public/index.html already exists. Overwrite? No
```

The create-react-app application creates a *build/* folder after we perform the `npm run build` for the first time. The folder contains all the merged content from the *public/* folder and the *src/* folder. Since it is a single page application, we want to redirect the user to the *index.html* file, so the React router can handle client-side routing.

Now your Firebase initialization is complete. This step created a few configuration files for Firebase Hosting in your project's folder. You can read more about them in [Firebase's documentation](https://firebase.google.com/docs/hosting)³¹⁰ for configuring redirects, a 404 page, or headers. Finally, deploy your React application with Firebase in the command line:

³¹⁰<https://bit.ly/3DVgpbG>

Command Line

```
firebase deploy
```

After a successful deployment, you should see a similar output with your project's identifier:

Command Line

```
Project Console: https://console.firebase.google.com/project/my-react-project-abc123\
/overview
```

```
Hosting URL: https://my-react-project-abc123.firebaseio.com
```

Visit both pages to observe the results. The first link navigates to your Firebase project's dashboard, where you'll see a new panel for the Firebase Hosting. The second link navigates to your deployed React application.

If you see a blank page for your deployed React application, make sure the `public` key/value pair in the `firebase.json` is set to `build`, or whichever name you chose for this folder. Second, verify you've run the build script for your React app with `npm run build`. Finally, check out the [official troubleshoot area for deploying create-react-app applications to Firebase](#)³¹¹. Try another deployment with `firebase deploy`.

Exercises:

- Read more about [Firebase Hosting](#)³¹².
- [Connect your domain to your Firebase deployed application](#)³¹³.
- Optional: If you want to have a managed cloud server, check out [DigitalOcean](#)³¹⁴. It's more work, but it allows more control. [I host all my websites, web applications, and backend APIs there](#)³¹⁵.
- Optional: [Rate this section](#)³¹⁶.

³¹¹<https://bit.ly/2ZaLMoE>

³¹²<https://bit.ly/3lXypAC>

³¹³<https://bit.ly/3phFxdp>

³¹⁴<https://m.do.co/c/fb27c90322f3>

³¹⁵<https://www.robinwieruch.de/deploy-applications-digital-ocean>

³¹⁶<https://forms.gle/QPjydK8UbaXkxCEj9>

Outline

We've reached the end of the Road to React, and I hope you enjoyed reading it, and that it helped you gain some traction in React. If you liked the book, share it with your friends who are interested in learning more about React. Also, a review on [Amazon](#)³¹⁷, [Google](#)³¹⁸, or [Goodreads](#)³¹⁹ would be very much appreciated.

From here, I recommend you extend the application to create your own React projects before engaging another book, course, or tutorial. Try it for a week, take it to production by deploying it, and reach out to me or others to showcase it. I am always interested in seeing what my readers built, and learning how I can help them along.

If you're looking for extensions for your application, I recommend several learning paths after you've mastered the basics:

- **Connecting to a Database and/or Authentication:** Growing React applications will eventually require persistent data. The data should be stored in a database so that keeps it intact after browser sessions, to be shared with different users. Firebase is one of the simplest ways to introduce a database without writing a backend application. In my book titled "[The Road to Firebase](#)"³²⁰, you will find a step-by-step guide on how to use Firebase authentication and database in React.
- **Connecting to a Backend:** React handles frontend applications, and we've only requested data from a third-party backend's API thus far. You can also introduce an API with a backend application that connects to a database and manages authentication/authorization. In "[The Road to GraphQL](#)"³²¹, I teach you how to use GraphQL for client-server communication. You'll learn how to connect your backend to a database, how to manage user sessions, and how to talk from a frontend to your backend application via a GraphQL API.
- **State Management:** You have used React to manage local component state exclusively in this learning experience. It's a good start for most applications, but there are also external state management solutions for React. I explore the most popular one in my book "[The Road to Redux](#)"³²².
- **Tooling with Webpack and Babel:** We used *create-react-app* to set up the application in this book. At some point you may want to learn the tooling around it, to create projects without *create-react-app*. I recommend a minimal setup with [Webpack](#)³²³, after which you can apply additional tooling.

³¹⁷<https://amzn.to/2JHIP42>

³¹⁸<https://books.google.de/books/about?id=RRLmDwAAQBAJ>

³¹⁹<https://www.goodreads.com/book/show/37503118-the-road-to-learn-react>

³²⁰<https://www.roadtofirebase.com/>

³²¹<https://www.roadtographql.com/>

³²²<https://www.roadtoredux.com/>

³²³<https://www.robinwieruch.de/minimal-react-webpack-babel-setup/>

- **Code Organization:** Recall the chapter about code organization and apply these changes, if you haven't already. It will help organize your components into structured files and folders, and it will help you understand the principles of code splitting, reusability, maintainability, and module API design. Your application will grow and need structured modules eventually; so it's better to start now.
- **Testing:** We only scratched the surface of testing. If you are unfamiliar with testing web applications, [dive deeper into unit testing and integration testing](#)³²⁴, especially with React applications. [Cypress](#)³²⁵ is a useful tool to explore for end-to-end testing in React.
- **Type Checking:** Earlier we used TypeScript in React, which is good practice to prevent bugs and improve the developer experience. Dive deeper into this topic to make your JavaScript applications more robust. Maybe you'll end up using TypeScript instead of JavaScript all along.
- **UI Components:** Many beginners introduce UI component libraries like Bootstrap too early in their projects. It is more practical to use a dropdown, checkbox, or dialog in React with standard HTML elements. Most of these components will manage their own local state. A checkbox has to know whether it is checked or unchecked, so you should implement them as controlled components. After you cover the basic implementations of these crucial UI components, introducing a UI component library should be easier.
- **Routing:** You can implement routing for your application with [react-router](#)³²⁶. There is only one page in the application we've created, but that will grow. React Router helps manage multiple pages across multiple URLs. When you introduce routing to your application, no requests are made to the web server for the next page. The router handles this client-side.
- **React Native:** [React Native](#)³²⁷ brings your application to mobile devices like iOS and Android. Once you've mastered React, the learning curve for React Native shouldn't be that steep, as they share the same principles. The only difference with mobile devices are the layout components, the build tools, and the APIs of your mobile device.

I invite you to visit my [website](#)³²⁸ to find more interesting topics about web development and software engineering. You can also [subscribe to my Newsletter](#)³²⁹ or [Twitter page](#)³³⁰ to get updates about articles, books, and courses.

Thank you for reading the Road to React.

Regards,

Robin Wieruch

³²⁴<https://www.robinwieruch.de/react-testing-tutorial>

³²⁵<https://www.robinwieruch.de/react-testing-cypress>

³²⁶<https://github.com/ReactTraining/react-router>

³²⁷<https://facebook.github.io/react-native/>

³²⁸<https://www.robinwieruch.de>

³²⁹<https://www.getrevue.co/profile/rwieruch>

³³⁰<https://twitter.com/rwieruch>