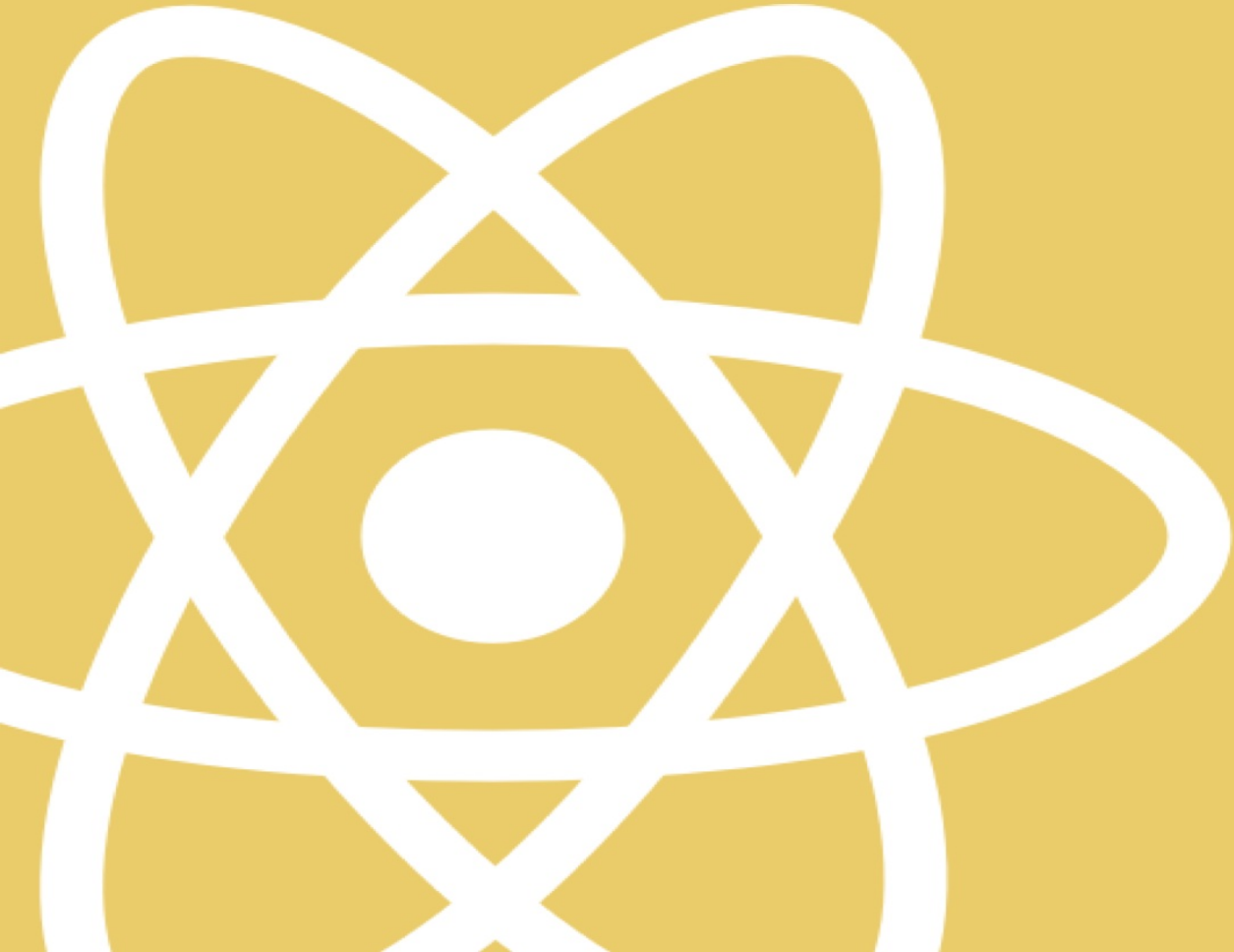
by robin wieruch

the Road to React



The Road to React

Your journey to master plain yet pragmatic React

Robin Wieruch

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

This version was published on 2020-04-20

* * * * *

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.

* * * * *

Table of Contents

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 for you. The book comes with additional referenced reading material and exercises with each chapter. After reading the book, you will be able to build your own applications in React. The material is kept up to date by myself and the community.

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

Essentially, you will learn to build your own React application from scratch, with features like pagination, client-side 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 engineer dedicated to learning and teaching programming in JavaScript. After obtaining my Master's Degree in computer science, I continued learning on my own. I gained experience from the startup world, where I used JavaScript intensively during both my professional life and spare time, which eventually led to a desire to teach others about these topics.

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

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

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

FAQ

How to get updates?

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

Also, if you have purchased a print 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, 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 take your suggestions into consideration. Don't expect any 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!

How to support the project?

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

What's your motivation behind the book?

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

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 about 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 this topic before continuing with the book. You will have lots of references to this fundamental knowledge in the 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 about the different aspects of programming. After picking up the fundamentals about 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 are 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 in 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 the first part of this learning experience, we'll cover the fundamentals of React, after which we'll create our first React project. Then we'll explore new aspects of React by implementing real features, the same as developing an actual web application. By the end we'll have a working React application with features like client and server-side searching, remote data fetching, and advanced state management.

Hello React

Single-page applications (SPA) 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 which 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 entire 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 was 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 *minimal HTML file* and *one associated JavaScript file*. After some network delay, the user sees the *by JavaScript rendered HTML* 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 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 :-)

Exercises

- Read more about why I moved from Angular to React.
- Read more about how to learn a framework.
- Read more about how to learn React.
- Read more about why frameworks matter.
- Scan through JavaScript fundamentals needed for React – without worrying too much about the React – to test yourself about several JavaScript features used in React.

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, as they will be covered thoroughly.

Editor and Terminal

I have provided a setup guide 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) or an IDE (e.g. Visual Studio Code). I recommend Visual Studio Code (also called VS Code) 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.

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 for the terms *editor*, *text editor*, and *IDE*, depending on what you decided to use for your setup.

Optionally, I recommend managing projects in GitHub while we conduct the exercises in this book, and I've provided a short guide 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.

If you don't want to set up the editor/terminal combination or IDE on your local machine, CodeSandbox, 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.

Node and NPM

Before we can begin, we'll need to have node and npm installed. Both are used to manage libraries (node packages) 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` command. If you don't receive output in the terminal indicating which version is installed, you need to install node and npm:

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 I created will get you up to speed.

Setting up a React Project

In the Road to React, we'll use create-react-app 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. In *create-react-app*, the tools and configurations 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:

- **README.md:** The *.md* extension indicates the file is a markdown file. Markdown is a lightweight markup language with plain text formatting syntax. Many source code 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 create-react-app GitHub repository.
- **node_modules/:** This folder contains all node packages that have been installed via npm. 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.

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 general 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:

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 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. Once you eject, you can't go back. Essentially this command is only there to make all the build tool 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 and getting started guide.
    - Read more about the supported JavaScript features in create-react-app.
- Read more about the folder structure in create-react-app.
    - Go through all of your React project's folders and files one by one.
- Read more about the scripts in create-react-app.
    - 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.
- 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.

Meet the React Component

Our first React component is in the *src/App.js* file, which should look similar to the example below. The file might differ slightly, because create-react-app will sometimes update the default component's structure.

src/App.js

```
import 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 from create-react-app.

src/App.js

```
import React from 'react';

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

export default App;
```

First, this React component, called App component, is just a JavaScript function. It's commonly called **function component**, because there are other variations of React components (see **component types** later). Second, the App component doesn't receive any parameters in its function signature yet (see **props** later). And third, the App component returns code that resembles HTML which is called JSX (see **JSX** later).

The function component possess implementation details like any other JavaScript function. You will see this in practice in action throughout your React journey:

src/App.js

```
import React from 'react';

function App() {
  // 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, like all JavaScript functions:

src/App.js

```
import React from 'react';

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

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

export default App;
```

Since we don't need anything from within the App component used for this variable – e.g. parameters coming from the function signature – we can define the variable outside of the App component as well:

src/App.js

```
import React from 'react';

const title = 'React';

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

export default App;
```

Let’s use this variable in the next section.

Exercises:

- Confirm your source code for the last section.
- 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.
    - Read more about const.
    - Read more about let.
- 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.

React JSX

Recall that I mentioned the returned output of the App component resembles HTML. This output is called JSX, which mixes HTML and JavaScript. Let's see how this works for displaying the variable:

src/App.js

```
import React from 'react';

const title = 'React';

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

export default App;
```

Start your application with `npm start` again, and look for the rendered variable in browser, which should read: "Hello React".

Let's focus on the HTML, which is expressed almost the same in JSX. An input field with a label can be defined as follows:

src/App.js

```
import 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>
  );
}

export default App;
```

We specified three HTML attributes here: `htmlFor`, `id`, and `type`. Where `id` and `type` should be familiar from native HTML, `htmlFor` might be new. The `htmlFor` reflects the `for` attribute in HTML. JSX replaces a handful of

internal HTML attributes, but you can find all the supported HTML attributes in React's documentation, which follow the camelCase 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. We have defined a string primitive to be displayed in the App component, and the same can be done with a JavaScript object:

src/App.js

```
import 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;
```

Remember, everything in curly braces in JSX can be used for JavaScript expressions (e.g. function execution):

src/App.js

```
import 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 is one of my favorite things about React. Without any extra templating syntax (except for the curly braces), we are now able to use JavaScript in HTML.

Exercises:

- Confirm your source code for the last section.
  - Confirm the changes from the last section.
- Read more about React's JSX.
- Define more primitive and complex JavaScript data types and render them in JSX.
- Try to render a JavaScript array in JSX. If it's too complicated, don't worry, because you will learn more about this in the next section.

Lists in React

So far we've rendered a few primitive variables in JSX; next we'll render a list of items. We'll experiment with sample data at first, then we'll apply that to fetch data from a remote API. First, let's define the array as a variable. As before, we can define a variable outside or inside the component. The following defines it outside:

src/App.js

```
import 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;
```

I used a `...` here as a placeholder, to keep my code snippet small (without App component 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 placeholder for code blocks that I have established previous exercises. 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. 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 */}
    </div>
  );
}
```

You can use the built-in JavaScript map method for arrays 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]
```

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

src/App.js

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

      <hr />

      {list.map(function(item) {
        return <div>{item.title}</div>;
      })}
    </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 in HTML.

React will display each item now, but you can still improve your code so React handles advanced dynamic lists more gracefully. By assigning a key attribute to each list item's element, React can identify modified items if the list changes (e.g. re-ordering). Fortunately, our list items come with an identifier:

src/App.js

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

      <hr />

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

Code Playground

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

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 />

      {list.map(function(item) {
```

```
        return (
          <div key={item.objectID}>
            <span>
              <a href={item.url}>{item.title}</a>
            </span>
            <span>{item.author}</span>
            <span>{item.num_comments}</span>
            <span>{item.points}</span>
          </div>
        );
      })}
    </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 anchor tag. Not only can JavaScript in JSX be used to display items, but also to assign HTML attributes dynamically.

Exercises:

- Confirm your source code for the last section.
  - Confirm the changes from the last section.
- Read more about why React's key attribute is needed (0, 1, 2). Don't worry if you don't understand the implementation yet, just focus on what problem it causes for dynamic lists.
- Recap the standard built-in array methods – especially *map*, *filter*, and *reduce* – which are available in native JavaScript.
- What happens if your return `null` instead of the JSX?
- Extend the list with some more items to make the example more realistic.
- Practice using different JavaScript expressions in JSX.

Meet another React Component

So far we've only been using the App component to create our applications. We used the App component in the last section to express everything needed to render our list in JSX, and it should scale with your needs and eventually handle more complex tasks. To help with this, we'll split some of its responsibilities into a standalone List component:

src/App.js

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

function App() { ... }

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

Optional: If this component looks odd, because the outermost part of the returned JSX starts with JavaScript. We could use it with a wrapping HTML element as well, but we'll continue with the previous version.

src/App.js

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

Now the new List component can be used in the App component:

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, we can see how components that encapsulate meaningful tasks can work for larger React applications.

Larger 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, so the List is a **child component** of the App. In a component tree, the App is the **root component**, and the components that don't render any other components are called **leaf components** (e.g. List). The App can have multiple children, as can the List. If the App has another child component, the additional child component is called a **sibling component** of the List.

Exercises:

- Confirm your source code for the last section.
  - Confirm the changes from the last section.
- Draw your components – the App component and List component – as a component tree on a sheet of paper. Extend this component tree with other possible components (e.g. extracted Search component for the input field and label in the App component). Try to figure out which other parts can be extracted as standalone components.
- If a Search component is used in the App component, would the Search component be a sibling, parent, or child component for the List component?
- Ask yourself what problems could arise if we keep treating the `list` variable as global variable. We will cover how to handle these problems in the upcoming sections.

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.

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"
```

If a JavaScript class definition exists, one can create *multiple* instances of it. It is similar to a React component, which 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>

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

      <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 like an HTML **element** anywhere in our JSX. The element produces an **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.

Exercises:

- Familiarize yourself with the terms *component definition*, *component instance*, and *element*.
- Experiment by creating multiple component instances of a List component.
- Think about how it could be possible to give each List component its own `list`.

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

import App from './App';

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

Next to React, there is another imported library called `react-dom`, in which a `ReactDOM.render()` function uses an HTML node to replace it with JSX. The process integrates React into HTML. `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.
- Consider how we can include a React application in an external web application that uses HTML.
- Read more about rendering elements in React.

React Component Definition (Advanced)

The following refactorings are optional recommendations 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.

All components in the *src/App.js* file are function component. 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
const (item) => { ... }

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

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

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

src/App.js

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

const List = () => {
  return list.map(function(item) {
```

```
    return (
      <div key={item.objectID}>
        ...
      </div>
    );
  });
};
```

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

src/App.js

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

If an arrow function doesn't do *anything* in between, but only returns *something*, – in other words, if an arrow function doesn't perform any task, but only returns information –, 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
count => {
  // perform any task in between

  return count + 1;
}

// with concise body
count =>
  count + 1;
```

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

src/App.js

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

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

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 for the last section.
  - Confirm the changes from the last section.
- Read more about JavaScript arrow functions.
- Familiarize yourself with arrow functions with block body and return, and concise body without return.

Handler Function in JSX

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

src/App.js

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

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

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

      <hr />

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

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

src/App.js

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

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

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

      <hr />

      <List />
    </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. This is called a **synthetic event** defined by a JavaScript object. Through this object, we can access the emitted value of the input field:

src/App.js

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

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

The synthetic event is essentially a wrapper around the browser's native event, 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.

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:

Code Playground

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

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

HTML and JavaScript work well together in JSX. JavaScript in HTML can display objects, can pass JavaScript primitives to HTML attributes (e.g. `href` to `<a>`), and can pass functions to an element's attributes for handling events.

I prefer using arrow functions because of their concision as event handlers, however, in a larger React component I see myself using the function

statements too, because it gives them more visibility in contrast to other variable declarations within a component's body.

Exercises:

- Confirm your source code for the last section.
  - Confirm the changes from the last section.
- Read more about React's events.

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 again in the List component. This could work if you only had one variable, but it doesn't scale with multiple variables across multiple components from many different files.

Using so called props, we can pass variables as information from one component to another component. Before using props, we'll move the list from the global scope into the App component and rename it to its actual domain:

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,
    },
  ];

  const handleChange = event => { ... };

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

Next, we'll use **React props** to pass the array to the List component:

src/App.js

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

  const handleChange = event => { ... };

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

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

      <hr />

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

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` attribute. We access it as `list` from the `props` object in the List component's function signature:

src/App.js

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

Using this operation, we've prevented the list/stories variable from polluting the global scope in the App component. Since `stories` is not used in the App component directly, but in one of its child components, we passed them as props to the List component. There, we can access it through the first function signature's argument, called `props`.

Exercises:

- Confirm your source code for the last section.
  - Confirm the changes from the last section.
- Read more about how to pass props to React components.

React State

React Props are used to pass information down the component tree; **React state** is used to make applications interactive. We'll be able to change the application's appearance by interacting with it.

First, there is a utility function called `useState` that we take from `React` for managing state. The `useState` function is called a hook. There is more than one **React hook** – related to state management but also other things in React – and you will learn about them throughout the next sections. For now, let's focus on **React's `useState` hook**:

src/App.js

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

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

  ...
};
```

React's `useState` hook takes an *initial state* as an argument. We'll use an empty string, and the function will return an array with two values. The first value (`searchTerm`) represents the *current state*; the second value is a *function to update this state* (`setSearchTerm`). I will sometimes refer to this function as *state updater function*.

If you are not familiar with the syntax of the two values from the returned array, consider reading about JavaScript array destructuring. It is used to read from an array more concisely. This is array destructuring and its benefits visualized in a nutshell:

Code Playground

```
// basic array definition
const list = ['a', 'b'];

// no array destructuring
const itemOne = list[0];
const itemTwo = list[1];

// array destructuring
const [firstItem, secondItem] = list;
```

In the case of React, the React `useState` hook is a function which returns an array. Take again the following JavaScript example as comparison:

Code Playground

```
function getAlphabet() {
  return ['a', 'b'];
}

// no array destructuring
const itemOne = getAlphabet()[0];
const itemTwo = getAlphabet()[1];

// array destructuring
const [firstItem, secondItem] = getAlphabet();
```

Array destructuring is just a shorthand version of accessing each item one by one. If you express it without the array destructuring in React, it becomes less readable:

src/App.js

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

  // less readable version without array destructuring
  const searchTermState = React.useState('');
  const searchTerm = searchTermState[0];
  const setSearchTerm = searchTermState[1];

  ...
};
```

The React team chose array destructuring because of its concise syntax and ability to name destructured variables. The following code snippet is an example of array destructuring:

src/App.js

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

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

  ...
};
```

After we initialize the state and have access to the current state and the state updater function, use them to display the current state and update it within the App component's event handler:

src/App.js

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

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

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

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

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

      <p>
        Searching for <strong>{searchTerm}</strong>.
      </p>

      <hr />

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

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

Exercises:

- Confirm your source code for the last section.
  - Confirm the changes from the last section.
- Read more about JavaScript array destructuring.
- Read more about React's useState Hook (0, 1), as it makes your React components interactive.

Callback Handlers in JSX

Next we'll focus on the input field and label, by separating a standalone Search component and creating an component instance of it in the App component. Through this process, the Search component becomes a sibling of the List component, and vice versa. We'll also move the handler and the state into the Search component to keep our functionality intact.

src/App.js

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

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

      <Search />

      <hr />

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

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>
  );
};
```

We have an extracted Search component that handles state and shows state without revealing its content. The component displays the `searchTerm` as text but doesn't share this information with its parent or sibling components yet. Since Search component does nothing except show the search term, it becomes useless for the other components.

There is no way to pass information as JavaScript data types up the component tree, since props are naturally only passed downwards. However, we can introduce a **callback handler** as a function: 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>

      <Search onSearch={handleSearch} />

      <hr />

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

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

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

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

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

Consider the concept of the callback handler: We pass a function from one component (App) to another component (Search); we call it in the second component (Search); but have the actual implementation of the function call in the first component (App). This way, we can communicate up the component tree. A handler function used in one component becomes a callback handler, which is passed down to components via React props. React props are always passed down as information 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 for the last section.
    - Confirm the changes from the last section.
- Revisit the concepts of handler and callback handler as many times as you need.

Lifting State in React

Currently, the Search component still has its internal state. While we established a callback handler to pass information up to the App component, we are not using it yet. We need to figure out how to share the Search component's state across multiple components.

The search term is needed in the App to filter the list before passing it to the List component as props. 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. The Search component doesn't manage the state anymore, but only passes up the event to the App component after text is entered into the input field. You could also display the `searchTerm` again in the App component or Search component by passing it down as prop.

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

By managing the search feature state in the App component, we can finally filter the list with the stateful `searchTerm` before passing the `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 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 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`.

src/App.js

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

  const searchedStories = stories.filter(function(story) {
    return 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. You have just added an interactive feature to your application.

The remaining section shows a couple of refactoring steps. We will be using the final refactored version in the end, so it makes sense to understand these steps and keep them. As learned before, we can make the function more concise using a JavaScript arrow function:

src/App.js

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

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

  ...
};
```

In addition, we could turn the return statement into an immediate return, because no other task (business logic) happens before the return:

src/App.js

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

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

  ...
};
```

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.

Now we can manipulate state in React, using the Search component's callback handler in the App component to update it. The current state is used as a filter for the list. With the callback handler, we used information from the Search component in the App component to update the shared state and indirectly in the List component for the filtered list.

Exercises:

- Confirm your source code for the last section.
  - Confirm the changes from the last section.

React Controlled Components

Controlled components are not necessary React components, but HTML elements. Here, we'll learn how to turn the Search component and its input field into a controlled component.

Let's go through a scenario that shows why we should follow the concept of controlled components throughout our React application. After applying the following change – giving the `searchTerm` an initial state – 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 according to the initial search, the input field doesn't show the initial `searchTerm`. We want the input field to reflect the actual `searchTerm` used from the initial state; but it's only reflected through the filtered list.

We need to convert the Search component with its input field into a controlled component. So far, the input field doesn't know anything about the `searchTerm`. It only uses the change event to inform us of a change. Actually, the input field has a `value` attribute.

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 starts with the correct initial value, using the `searchTerm` from the React state. Also, when we change the `searchTerm`, we force the input field to use the value from React's state (via props). Before, the input field managed its own internal state natively with just HTML.

We learned about controlled components in this section, and, taking all the previous sections as learning steps into consideration, 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 be passed down as props to other components. It's rendered for the first time as a UI. Once a side-effect occurs, like user input or data loading from a remote API, the change is captured in React's state. Once state has been changed, all the components affected by the modified state or the implicitly modified props are re-rendered (the component functions runs 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 state is changed (e.g. current state changed via state updater function from `useState`), all components affected by modified state/props render again.

Every run through a component's function takes the *recent value* (e.g. current state) from the hooks 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 for the last section.
    - Confirm the changes from the last section.
- Read more about controlled components in React.
- Experiment with `console.log()` in your React components and observe how your changes render, both initially and after the input field changes.

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: The following refactorings are recommended for you to learn different JavaScript/React patterns, though you can still build complete React applications without them. Consider this advanced React techniques that will make your source code more concise.

React props are a JavaScript object, else we couldn't access `props.list` or `props.onSearch` in 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:

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 concise body into block body:

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 function signature, we can conveniently access all information without dealing with its `props` container. This should be the basic lesson learned from this section, however, we can take this one step further with the following advanced lessons.

Let's check out another scenario to dive deeper into advanced props handling in React: In order to prepare for this scenario, we will extract a new Item component from the List component with the previous lesson learned about object destructuring for React's `props` object:

src/App.js

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

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

Now, the incoming `item` in the Item component has something in common with the previously discussed `props`: 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.

The shown 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 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"
```

Nested destructuring helps us to access properties from objects which are deeply nested (e.g. the pet's name of a user). Now, in our Item components, because the `item` object is never directly used in the Item component's JSX elements, we can perform a *nested destructuring* in the component's function signature too:

src/App.js

```
// Variation 1: Nested Destructuring

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

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

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.

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 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. Iteration

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

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

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:

Code Playground

```
const profile = {
  firstName: 'Robin',
```

```
  lastName: 'Wieruch',
};

const address = {
  country: 'Germany',
  city: 'Berlin',
  code: '10439',
};

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

console.log(user);
// {
//   firstName: "Robin",
//   lastName: "Wieruch",
//   gender: "male"
//   country: "Germany,
//   city: "Berlin",
//   code: "10439"
// }
```

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. Iteration

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

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

This refactoring made the process of passing the information from List to Item component more concise. Finally, we'll use JavaScript's rest parameters 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 right side of an assignment, the spread operator happens on the left 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 `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)

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

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

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 used commonly 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 will find more likely 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 the next chapters:

src/App.js

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

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

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 make sure they're using a common React code style.

Exercises:

- Confirm your source code for the last section.
    - Confirm the changes from the last section.
- Read more about JavaScript's destructuring assignment.
- 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.
- Read more about JavaScript's rest parameters.
- Get a good sense about JavaScript (e.g. spread operator, rest parameters, destructuring) and what's related to React (e.g. props) from the last lessons.
- Continue to use your favorite way to handle React's props. If you're still undecided, consider the variation used in the previous section.

React Side-Effects

Next we'll add a feature to our Search component in the form of another React hook. We'll make the Search component remember the most recent search interaction, so the application opens it in the browser whenever it restarts.

First, use the local storage of the browser to store the `searchTerm` accompanied by an identifier. Next, use the stored value, if there a value exists, to set the initial state of the `searchTerm`. Otherwise, the initial state defaults to our initial state (here "React") as before:

src/App.js

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

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

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

    localStorage.setItem('search', event.target.value);
  };

  ...
);
```

When using the input field and refreshing the browser tab, the browser should remember the latest search term. Using the local storage in React can be seen as a **side-effect** because we interact outside of React's domain by using the browser's API.

There is one flaw, though. The handler function should mostly be concerned about updating the state, but now it has a side-effect. If we use the `setSearchTerm` function elsewhere in our application, we will break the feature we implemented because we can't be sure the local storage will also get updated. Let's fix this by handling the side-effect at a dedicated 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);
  };

  ...
);
```

React's useEffect Hook takes two arguments: The first argument is a function where the side-effect occurs. In our case, the side-effect is when the user types the `searchTerm` into the browser's local storage. The optional second argument is a dependency array of variables. If one of theses variables changes, the function for the side-effect is called. In our case, the function is called every time the `searchTerm` changes; and it's also called initially when the component renders for the first time.

Leaving out the second argument, to be specific 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. The hook lets us opt into React's component lifecycle. It can be triggered when the component is first mounted, but also one of its dependencies are updated.

Using React `useEffect` instead of managing the side-effect in the handler has made the application more robust. *Whenever* and *wherever* `searchTerm` is updated via `setSearchTerm`, local storage will always be in sync with it.

Exercises:

- Confirm your source code for the last section.
  - Confirm the changes from the last section.
- Read more about React's useEffect Hook (0, 1).

- 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.

React Custom Hooks (Advanced)

Thus far we've covered the two most popular hooks in React: useState and useEffect. useState is used to make your application interactive; useEffect is used to opt into the lifecycle of your components.

We'll eventually cover more hooks that come with React – in this volume and in other resources – though we won't cover all of them here. 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. Start by extracting all relevant implementation details from the App component into this 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, it's just a function around our previously in the App component used `useState` and `useEffect` hooks. Before we can use it, let's return the values that are needed in our App component from this custom hook:

src/App.js

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

  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. Now we can use the custom hook with its returned values in the App component with the usual array destructuring:

src/App.js

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

  const [searchTerm, setSearchTerm] = useSemiPersistentState();

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

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

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

Another goal of a custom hook should be reusability. All of this custom hook's internals are about the search domain, but the hook should be for a value that's set in state and synchronized in local storage. Let's adjust the naming therefore:

src/App.js

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

  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. To fix this, pass in a key:

src/App.js

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

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

  return [value, setValue];
};

const App = () => {
  ...

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

  ...
};
```

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. Without it, the side-effect may run with an outdated key (also called *stale*) if the key changed between renders.

Another improvement is to give the custom hook the initial state we had from the outside:

src/App.js

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

  ...
};

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; 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 for the last section.
  - Confirm the changes from the last section.
- Read more about React Hooks to get a good understanding of them. They are the bread and butter in React function components, so it's important to really understand them (0, 1).

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 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. Both Search elements should be visible in your browser now, with input field and label. So if you prefer to omit the wrapping `<div>` or `<span>` elements, substitute them with an empty tag that is allowed in JSX, and doesn't introduce intermediate elements in our rendered HTML.

Exercises:

- Confirm your source code for the last section.
  - Confirm the changes from the last section.
- Read more about React fragments.

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. It also 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.

Since the Search component doesn't have any actual "search" functionality, it takes little effort to generalize other search domain properties to make the component reusable for the rest of the application. Let's pass an additional `id` and `label` prop to the Search component, rename the actual value and callback handler to something more abstract, and rename the component accordingly:

src/App.js

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

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

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

      ...
    </div>
  );
};

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

From the App component, no `type` prop is passed to the InputWithLabel component, so it is not specified from the outside. The default parameter from the function signature takes over for the input field.

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 for the last section.
  - Confirm the changes from the last section.
- Read more about Reusable React Components.
- 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.

React Component Composition

Now 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}
        onInputChange={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. 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',
  onInputChange,
  children,
}) => (
  <>
    <label htmlFor={id}>{children}</label>
     
    <input
      id={id}
      type={type}
      value={value}
      onChange={onInputChange}
    />
  </>
);
```

Now the React component's elements behave similar to native HTML. Everything that's passed between a component's elements can be accessed as `children` in the component and be rendered somewhere. Sometimes when using a React component, you want to have more freedom from the outside what to render in the inside of a component:

src/App.js

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

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

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

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

With this React feature, we can compose React components into each other. We've used it with a JavaScript string and with a string wrapped in an HTML `<strong>` element, but it doesn't end here. You can pass components via React children as well.

Exercises:

- Confirm your source code for the last section.
  - Confirm the changes from the last section.
- Read more about React Component Composition (0, 1).
- Create a simple text component that renders a string and passes it as `children` to the InputWithLabel component.

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, in cases such as these:

- read/write access to elements via the DOM API:
  - measure (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:
  - D3 is a popular imperative chart library

Because imperative programming in React is often verbose and counterintuitive, we'll walk only through 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>
     
    <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 auto-focus 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 an autofocus or not:

src/App.js

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

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

      <InputWithLabel
        id="search"
        value={searchTerm}
        isFocused
        onInputChange={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',
  onInputChange,
  isFocused,
  children,
}) => (
  <>
    <label htmlFor={id}>{children}</label>
     
    <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'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>
       
       {/* 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 whenever it's necessary. This lesson is for the sake of learning about the DOM API in React.

Exercises:

- Confirm your source code for the last section.
  - Confirm the changes from the last section.
- Read more about React's useRef Hook.

Inline Handler in JSX

The list of stories we have so far is only an unstateful variable. We can filter the rendered list with the search feature, but the list itself stays intact if we remove the filter. The filter is just a temporary change through a third party, 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 are the current state (`stories`) and the state updater function (`setStories`). We aren't using the custom `useSemiPersistentState` hook yet, because we don't want to open the browser with the cached list each time. Instead, we always want to start with the initial list.

src/App.js

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

const useSemiPersistentState = (key, initialState) => { ... };

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

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

  ...
};
```

The application behaves the same because the `stories`, now returned from `useState`, are still filtered into `searchedStories` and displayed in the List. 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 receives an item to be removed as an argument, and filters the current stories based on this information by removing all items that don't meet its condition(s). The returned stories are then set as new state, and the List component passes the function to its child component. It's not using this new information; it's just passing it on:

src/App.js

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

Finally, we can use the incoming function in another handler in the Item component to pass the `item` to it. A button element is used to trigger the actual event:

src/App.js

```
const Item = ({ item, onRemoveItem }) => {
  const handleRemoveItem = () => {
    onRemoveItem(item);
  };

  return (
    <div>
      <span>
        <a href={item.url}>{item.title}</a>
      </span>
      <span>{item.author}</span>
```

```
      <span>{item.num_comments}</span>
      <span>{item.points}</span>
      <span>
        <button type="button" onClick={handleRemoveItem}>
          Dismiss
        </button>
      </span>
    </div>
  );
};
```

We could have passed only the item's `objectID`, since that's all we need in the App component's callback handler, but we aren't sure what information the handler might need later. It may need more than an identifier to remove an item. If we call the handler `onRemoveItem`, it should be the item being passed, not just its identifier.

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. Since a handler is just a function, and in this case it doesn't return anything, we could remove the block body for it for the sake of completeness.

src/App.js

```
const Item = ({ item, onRemoveItem }) => {
  const handleRemoveItem = () =>
    onRemoveItem(item);

  ...
};
```

This change makes our source code less readable as we accumulate handlers in the function component. Sometimes I refactor handlers in a function component from an arrow function back to a normal function statement, just to make the component more explorable:

src/App.js

```
const Item = ({ item, onRemoveItem }) => {
  function handleRemoveItem() {
    onRemoveItem(item);
  }

  ...
};
```

In this section we applied props, handlers, callback handlers, and state. That are all lessons learned from before. Now we'll tackle **inline handlers**, which allow us to execute the function right in the JSX. There are two solutions using the incoming function in the Item component as an inline handler. First, using JavaScript's bind method:

src/App.js

```
const Item = ({ item, onRemoveItem }) => (
  <div>
    <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>
  </div>
);
```

Using JavaScript's bind method 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.

The second and more popular solution is to use a wrapping arrow function, which allows us to sneak in arguments like `item`:

src/App.js

```
const Item = ({ item, onRemoveItem }) => (
  <div>
    <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>
  </div>
);
```

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 }) => (
  <div>
    ...
    <span>
      <button
        type="button"
        onClick={() => {
          // do something else

          // note: avoid using complex logic in JSX

          onRemoveItem(item);
        }}
      >
        Dismiss
      </button>
    </span>
  </div>
);
```

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; the inline handler move the implementation details into the JSX.

Exercises:

- Confirm your source code for the last section.
  - Confirm the changes from the last section.
- Review handlers, callback handlers, and inline handlers.

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.

Sometimes we must render a component before we can fetch data from a third-party API and display it. In the following, we will simulate this kind of asynchronous data in the application. In a live application, this asynchronous data gets fetched from a real remote API. We start off with a function that returns a promise – data, once it resolves – in its shorthand version. 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. 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 an 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 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 for the last section.
  - Confirm the changes from the last section.
- Read more about JavaScript Promises.

React Conditional Rendering

Handling asynchronous data in React leaves us with conditional states: with data and without data. This case is already covered, though, because our initial state is an empty list rather than `null`. If it was null, we'd have to handle this issue in our JSX. However, since it's `[]`, we `filter()` over an empty array for the search feature, which leaves us with an empty array. This leads to rendering nothing in the List component's `map()` function.

In a real world application, there are more than two conditional states for asynchronous data, though. Consider showing users a loading indicator when data loading is delayed:

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, we can inline this conditional state as a **conditional rendering** in JSX:

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 solve 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 went 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.

This feature 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 for the last section.
  - Confirm the changes from the last section.
- Read more about conditional rendering in React.

React Advanced State

All state management in this application makes heavy use of React's useState Hook. More sophisticated state management gives you **React's useReducer Hook**, though. 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 often 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 returned from `useState`. Instead of setting 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 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 for the last section.
  - Confirm the changes from the last section.
- Read more about reducers in JavaScript.
- Read more about reducers and useReducer in React (0, 1).

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);

  ...
};
```

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 }
  );

  ...
};
```

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, we must handle them 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`, but sets the `isError` boolean flags yet keeps all the other state intact (e.g. `stories`). That's how we get around the bug introduced earlier, since an error should remove the loading state.

Observe how the `REMOVE_STORY` action changed as well. It operates on the `state.data` , not longer just 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 for the last section.
  - Confirm the changes from the last section.
- Read over the previously linked tutorials about reducers in JavaScript and React.
- Read more about when to use useState or useReducer in React.

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 third-party API. We will use the reliable and informative Hacker News API 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 topic). In this case, we fetch stories about React (B). Second, the native browser's fetch API 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.

In the previous code example we used JavaScript's Template Literals 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. For the App component, there wasn't much data fetching to implement here, though it's all part of learning how to manage asynchronous data as state in React.

Exercises:

- Confirm your source code for the last section.
  - Confirm the changes from the last section.
- Read through Hacker News and its API.
- Read more about the browser native fetch API for connecting to remote APIs.
- Read more about JavaScript's Template Literals.

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, so we'll implement data refetching. If the `searchTerm` changes, run the side-effect for the data fetching again. 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 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, 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 for the last section.

- Confirm the changes from the last section.

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(() => {
    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 with each App component is rendered. The `handleFetchStories` function would be created each time, 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
...
```

The `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 for the last section.
  - Confirm the changes from the last section.
- Read more about React's useCallback Hook.

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 are out of our reach. Eventually, we will incur rate limiting, 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
        onInputChange={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` 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' })
      );
  }, [url]);

  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. Too many responsibilities one may would have said. Now it's only used for the former. A second state called `url` 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 for the last section.
  - Confirm the changes from the last section.
- Why is `useState` instead of `useSemiPersistentState` used for the `url` state management?
- Why is there no check for an empty `searchTerm` in the `handleFetchStories` function anymore?

Third-Party Libraries in React

We previously introduced the native fetch API to complete requests to the Hacker News API, which the browser provides. Not all browsers support this, however, especially the older ones. Also, once you start testing your application in a headless browser environment, issues can arise with the fetch API. There are a couple of ways to make fetch work in older browsers (polyfills) 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 axios, 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 on the command line:

Command Line

```
npm install axios
```

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

src/App.js

```
import 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 => {
```

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

  ...
};
```

In this code, we call axios `axios.get()` for an explicit HTTP GET request, 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 over the native fetch API when requests become complex, working with older browser, or for testing.

Exercises:

- Confirm your source code for the last section.
  - Confirm the changes from the last section.
- Read more about popular libraries in React.
- Read more about why frameworks matter.
- Read more about axios.

Async/Await in React (Advanced)

You'll work with asynchronous data often in React, so it's good to know 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, everything reads like synchronous code. Actions after the `await` keyword are not executed until promise resolves, meaning the code will wait.

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]);

  ...
};
```

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.

Exercises:

- Confirm your source code for the last section.
  - Confirm the changes from the last section.
- Read more about data fetching in React.
- Read more about async/await in JavaScript.

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
          onInputChange={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. In the next two steps, we will only separate the component into its standalone SearchForm component:

src/App.js

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

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

The new component is used by the App component. 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 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`.

Exercises:

- Confirm your source code for the last section.
  - Confirm the changes from the last section.
- Try the code without `preventDefault`.
- Read more about preventDefault for Events in React.
- Read more about React Component Composition.

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.

I won't cover all that's considered legacy in React, because some features have been revamped more than once. You may see the previous iteration of the feature in older React applications, but will probably be different than the current.

Throughout this section we will compare a modern React application to its legacy version. We'll discover that most differences between modern and legacy React are due to class components versus function components.

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.

Code Playground

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

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

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) all React's component features (e.g. state management for state, lifecycle methods for side-effects). React props are accessed via the class instance (`this`).

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:

Code Playground

```
const InputWithLabel = ({
  id,
  value,
  type = 'text',
  onInputChange,
```

```
  children,
}) => (
  <>
    <label htmlFor={id}>{children}</label>
     
    <input
      id={id}
      type={type}
      value={value}
      onChange={onInputChange}
    />
  </>
);
```

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 the more lightweight function component.

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

Exercises:

- Read more about JavaScript Classes.
- Read more about how to refactor from a class component to a function component.
- Learn more about a different class component syntax which wasn't popular but more effective.
- Read more about class components in depth.
- Read more about other legacy and modern component types in React.

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).

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
- createRef Refs (exclusive for Class Components)
- useRef Hook Refs (exclusive for Function Components)

Recently, the React team introduced **React's createRef** as the latest equivalent to a function component's useRef hook:

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` attributed, and here used in a lifecycle method. 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 its 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.

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, 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.

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), they might need to be configured to import CSS or SVG files. Since we are using create-react-app, we can use these files as assets right away.

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 in a CSS file later.

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 React from 'react';
import axios from 'axios';

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

This CSS file will define the two (and more) CSS classes we used 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 }) => (
  <div 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>
  </div>
);
```

As you can see, we can also use the native `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, as a separate style definition keeps the JSX more concise.

In your *src/App.css* file, define the new CSS classes. Basic CSS features are used. Advanced CSS features (e.g. nesting) from CSS extensions (e.g. Sass) are not included in this example, as they are optional configurations.

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 specific button styles (small and large). One of the button specifications has been used, the other will be used in the next steps.

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) 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
    </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 from a CSS file. Lastly, style the InputWithLabel component:

src/App.js

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

  return (
    <>
      <label htmlFor={id} className="label">
        {children}
      </label>
       
      <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.

This is the basic CSS most of us have already learned, written with an inline style that is more dynamic. Without CSS extensions like Sass (Syntactically Awesome Style Sheets) inline styles can become burdensome, though, because features like CSS nesting are not available in native CSS.

Exercises:

- Confirm your source code for the last section.
  - Confirm the changes from the last section.
- Read more about CSS stylesheets in create-react-app.
- Read more about Sass in create-react-app for taking advantage of more advanced CSS features like nesting.
- 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.

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 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 assigns 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 }) => (
  <div 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>
  </div>
);
```

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. 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 library, which is installed via the command line as project dependency:

src/App.js

```
import cs from 'classnames';

...

// somewhere in a className attribute
className={cs(styles.button, styles.buttonLarge)}
```

The library offers conditional stylings as well. Finally, continue with the InputWithLabel component:

src/App.js

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

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

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 applies as the last section: 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 the JavaScript each time a style isn't defined. In the standard CSS approach, unmatched styles in the JavaScript and CSS files might go unnoticed.

Exercises:

- Confirm your source code for the last section.
  - Confirm the changes from the last section.
- Read more about CSS Modules in create-react-app.

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 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.div`
  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 elements (`form`, `button`) are 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
      onInputChange={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>
       
      <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 style 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 for the last section.
  - Confirm the changes from the last section.
- Read more about Styled Components in React.
  - 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.

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. 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 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\
25.023,27.332c1.9-1.901,4.48-2.97,7.168-2.97l0,0
      c2.688,0,5.268,1.068,7.167,2.97l54.675,54.701c3.956,3.957,3.956,10.372,0,14.32\
8L120.803,269.668
      C118.903,271.57,116.325,272.638,113.636,272.638z M24.463,158.958l89.173,89.209\
l158.9-158.97l-40.346-40.364L120.803,160.264
      c-1.9,1.902-4.478,2.971-7.167,2.971c-2.688,0-5.267-1.068-7.168-2.97l-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:

src/App.js

```
import 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 as a `height` and `width` attribute:

src/App.js

```
const Item = ({ item, onRemoveItem }) => (
  <div 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>
  </div>
);
```

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 for the last section.
  - Confirm the changes from the last section.
- Read more about SVGs in create-react-app.
- Read more about SVG background patterns in React.
- Add another SVG icon in your application.

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 can 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, 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 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 this 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 a first time render of the component using this custom hook. It doesn't make sense to run the side-effect for the initial rendering of the component, 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) 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` 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.

Note: This technique isn't only used for performance optimizations, but for the sake of having a side-effect run only when a component re-renders. I used it several times, and I suspect you'll stumble on one or the other use case for it eventually.

Don't re-render if not needed

Earlier, we 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') ||
  list.map(item => (
    <Item
      key={item.objectID}
      item={item}
      onRemoveItem={onRemoveItem}
    />
  ));
```

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
```

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
```

But the List component shouldn’t re-render. The search feature isn’t executed via its button, so the `list` passed to the List component should remain the same. This is React’s default behavior, which surprises many people.

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 still fast.

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') ||
    list.map(item => (
      <Item
        key={item.objectID}
        item={item}
        onRemoveItem={onRemoveItem}
      />
    ))
);
```

However, the output stays the same when typing into the SearchForm's input field:

Visualization

```
B:App
B:List
```

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. Earlier, we used React's useCallacbk 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 `memo` and `useCallback`, to search via the SearchForm's input field. The "B:List" output disappears, and only the App component re-renders with its "B:App" output.

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 fast enough. However, if re-rendering decreases the performance of a React application, `memo` helps prevent re-rendering.

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.

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. It prevents creating the function on every render, so the `useCallback` hook becomes unnecessary. 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 optimization only if you run into a 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 for the last section.
  - Confirm the changes from the last section.
- Read more about React's memo API.
- Read more about React's useCallback Hook.
- 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. Use this section as reference, in case you run into performance problems.

TypeScript in React

TypeScript for JavaScript and React have 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 of JavaScript development. While it improves the developer experience, the code also 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 code base 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 Typescript 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-app:

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 safety for the entire *src/App.tsx* file, starting with typing the arguments of the custom hook:

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 `state updater function` that take 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 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;
}) => (
  <div>
    ...
  </div>
);
```

There are two problems: the code is verbose, and it has duplications. 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;
}) => (
  <div>
```

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

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 Item component outside:

src/App.tsx

```
type ItemProps = {
  item: Story;
  onRemoveItem: (item: Story) => void;
};

const Item = ({ item, onRemoveItem }: ItemProps) => (
  <div>
    ...
  </div>
);
```

That's the most popular way to type React component's props with TypeScript. 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) =>
  list.map(item => (
    <Item
      key={item.objectID}
      item={item}
      onRemoveItem={onRemoveItem}
    />
  ));
```

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.

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, except without looking into the `state` and `action` types. As the application's developer, we know both objects and their shapes 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 stories state, the current state, and the action are types; the return new 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 an 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.

There is still a type safety issue in the App component's return statement for the returned List component. It can be fixed by giving the List component a wrapping HTML `div` element or a React fragment:

src/App.tsx

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

```
  </>
);
```

According to a TypeScript with React issue on GitHub: *"This is because due to limitations in the compiler, function components cannot return anything other than a JSX expression or null, otherwise it complains with a cryptic error message saying that the other type is not assignable to Element."*

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** signaled with a question mark:

src/App.tsx

```
type InputWithLabelProps = {
  id: string;
  value: string;
  type?: string;
  onInputChange: (event: React.ChangeEvent<HTMLInputElement>) => void;
  isFocused?: boolean;
  children: React.ReactNode;
};

const InputWithLabel = ({
  id,
  value,
  type = 'text',
  onInputChange,
  isFocused,
  children,
}: InputWithLabelProps) => {
  ...
};
```

Both the `type` and `isFocused` properties are optional. Using TypeScript, you can tell the compiler 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 for the last section.
  - Confirm the changes from the last section.
- Dig into the React + TypeScript Cheatsheet, 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 off 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.

Unit Testing to Integration Testing

Testing source code is essential to programming, and should be seen as a mandatory exercise for serious developers. We want to verify our source code's quality and functionality before using it in production. The testing pyramid will serve as our guide.

The testing pyramid includes end-to-end tests, integration tests, and unit tests. Unit tests are used for small, isolated blocks of code, such as a single function or component. Integration tests help us figure out if these units work well together. An end-to-end test simulates a real-life scenario, such as the login flow in a web application. Unit tests are quick and easy to write and maintain; end-to-end tests are the opposite.

We want to have many unit tests covering our functions and components. After that, we can use several integration tests to make sure the most important functions and components work together as expected. Finally, we may need a few end-to-end tests to simulate critical scenarios. In this learning experience, we will cover **unit and integration tests**, along with a component specific testing technique called **snapshot tests**. **E2E tests** will be part of the exercise.

Since there are many testing libraries, it can be challenging to choose one as a beginner to React. We will use Jest by Facebook as a testing framework to avoid making this tutorial too opinionated. Most of the other testing libraries for React use Jest as foundation, so it's a good introduction.

Unit to Integration Testing

Often the lines between unit and integration tests are unclear. Testing the List component with its Item component could be considered an integration test, but it could also be a unit test for two tightly coupled components. In this section, we start with unit testing and move towards integration testing. Everything in between is a spectrum between both.

Let's start with a pseudo test in your *src/App.test.js* file:

src/App.test.js

```
describe('something truthy', () => {
  it('true to be true', () => {
    expect(true).toBe(true);
  });
});
```

Fortunately, create-react-app comes with Jest. You can run the test using the interactive create-react-app test script on the command line. The output for all test cases will be presented in your command line interface.

Command Line

```
npm test
```

Jest matches all files with a *test.js* suffix in its filename when its command is run. Successful tests are displayed in green; failed tests are displayed in red:

src/App.test.js

```
describe('something truthy', () => {
  it('true to be true', () => {
    expect(true).toBe(false);
  });
});
```

Tests in Jest consist of **test suites** (`describe`), which are comprised of **test cases** (`it`), which have **assertions** (`expect`) that turn out green or red:

src/App.test.js

```
// test suite
describe('truthy and falsy', () => {
  // test case
  it('true to be true', () => {
    // test assertion
    expect(true).toBe(true);
  });

  // test case
  it('false to be false', () => {
    // test assertion
    expect(false).toBe(false);
  });
});
```

The “it”-block describes one test case. It comes with a test description that returns success or failure. We can also wrap this block into a “describe”-block that defines our test suite with many “it”-blocks for one specific component. Both blocks are used to organize your test cases. Note that the

`it` function is known in the JavaScript community as a single-test case function; in Jest, however, `it` is often used as an alias `test` function.

src/App.test.js

```
describe('something truthy', () => {
  test('true to be true', () => {
    expect(true).toBe(false);
  });
});
```

To use React components in Jest, we require a utility library for rendering components in a test environment:

Command Line

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

Also, before you can test your first components, you have to export them from your *src/App.js* file:

src/App.js

```
...

export default App;

export { SearchForm, InputWithLabel, List, Item };
```

Import them along with the previously installed utility library in the *src/App.test.js* file:

src/App.test.js

```
import React from 'react';
import renderer from 'react-test-renderer';

import App, { Item, List, SearchForm, InputWithLabel } from './App';
```

Write your first component test for the Item component. The test case renders the component with a given `item` using the utility library:

src/App.test.js

```
import React from 'react';
import renderer from 'react-test-renderer';

import App, { Item, List, SearchForm, InputWithLabel } from './App';

describe('Item', () => {
  const item = {
    title: 'React',
    url: 'https://reactjs.org/',
```

```
    author: 'Jordan Walke',
    num_comments: 3,
    points: 4,
    objectID: 0,
  };

  it('renders all properties', () => {
    const component = renderer.create(<Item item={item} />);

    expect(component.root.findByType('a').props.href).toEqual(
      'https://reactjs.org/'
    );
  });
});
```

Information about a component's or element's attributes are available via the `props` property. In the test assertion, we find the anchor tag (`a`) and its `href` attribute, and perform an equality check. If the test turns out green, we can be sure the anchor tag's `href` attribute is set to the correct `url` property of the `item`. In the same test case, we can add more test assertions for the other item's properties:

src/App.test.js

```
describe('Item', () => {
  const item = {
    title: 'React',
    url: 'https://reactjs.org/',
    author: 'Jordan Walke',
    num_comments: 3,
    points: 4,
    objectID: 0,
  };

  it('renders all properties', () => {
    const component = renderer.create(<Item item={item} />);

    expect(component.root.findByType('a').props.href).toEqual(
      'https://reactjs.org/'
    );

    expect(
      component.root.findAllByType('span')[1].props.children
    ).toEqual('Jordan Walke');
  });
});
```

Since there are multiple `span` elements, we find all of them and select the second one (index is `1` , because we count from `0`) and compare its React `children` prop to the item's `author`. This test isn't thorough enough, though. Once the order of `span` elements in the Item component changes, the test fails. Avoid this flaw by changing the assertion to:

src/App.test.js

```
describe('Item', () => {
  const item = { ... };

  it('renders all properties', () => {
    ...

    expect(
      component.root.findAllByProps({ children: 'Jordan Walke' })
        .length
    ).toEqual(1);
  });
});
```

The test assertion isn't as specific anymore. It just tests whether there is one element with the item's `author` property. You can apply this technique for all the other properties of the item yourself. Otherwise, leave it for later as exercise.

We tested whether the Item component renders as text or HTML attributes (`href`), but we didn't test the callback handler. The following test case makes this assertion by simulating a click event via the `button` element's `onClick` attribute:

src/App.test.js

```
describe('Item', () => {
  const item = { ... };

  it('renders all properties', () => {
    ...
  });

  it('calls onRemoveItem on button click', () => {
    const handleRemoveItem = jest.fn();

    const component = renderer.create(
      <Item item={item} onRemoveItem={handleRemoveItem} />
    );

    component.root.findByType('button').props.onClick();

    expect(handleRemoveItem).toHaveBeenCalledTimes(1);
    expect(handleRemoveItem).toHaveBeenCalledWith(item);

    expect(component.root.findAllByType(Item).length).toEqual(1);
  });
});
```

Jest lets us pass a test-specific function to the Item component as 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.

Item component's unit test is complete, because we tested input (`item`) and output (`onRemoveItem`). The two shouldn't be confused with input (arguments) and output (JSX) of the function component, which were also tested as. One last improvement makes the test suite for the Item component more concise by giving it a shared setup function:

src/App.test.js

```
describe('Item', () => {
  const item = { ... };
  const handleRemoveItem = jest.fn();

  let component;

  beforeEach(() => {
    component = renderer.create(
      <Item item={item} onRemoveItem={handleRemoveItem} />
    );
  });

  it('renders all properties', () => {
    expect(component.root.findByType('a').props.href).toEqual(
      'https://reactjs.org/'
    );

    ...
  });

  it('calls onRemoveItem on button click', () => {
    component.root.findByType('button').props.onClick();

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

A common setup (or teardown) function in tests removes duplicated code. Since the component must be rendered for both test cases, and the props are the same for both renderings, we can share this code in a common setup function. From there, we'll move on to testing the List component:

src/App.test.js

```
...

describe('Item', () => {
  ...
```

```
});

describe('List', () => {
  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,
    },
  ];

  it('renders two items', () => {
    const component = renderer.create(<List list={list} />);

    expect(component.root.findAllByType(Item).length).toEqual(2);
  });
});
```

The test checks straightforward whether two Item components are rendered for the two items in the list. You could continue testing the List component by checking whether each callback handler (`onRemoveItem`) is called for each Item component, which would have a similar solution to the previous Item component's test. Is this test still a unit test or already an integration test?

Keeping this question in the room, we will move on to the SearchForm with InputWithLabel component:

src/App.test.js

```
describe('SearchForm', () => {
  const searchFormProps = {
    searchTerm: 'React',
    onSearchInput: jest.fn(),
    onSearchSubmit: jest.fn(),
  };

  let component;

  beforeEach(() => {
    component = renderer.create(<SearchForm {...searchFormProps} />);
  });

  it('renders the input field with its value', () => {
```

```
    const value = component.root.findByType(InputWithLabel).props
      .value;

    expect(value).toEqual('React');
  });
});
```

In this test, we assert whether the InputWithLabel component receives the correct prop from the SearchForm component. Essentially the test stops before the InputWithLabel component, because it only tests the interface (props) of it. Arguably it's still a unit test, because the underlying implementation details of the InputWithLabel component could change without changing the interface. You can change the test to make it work through to the InputWithLabel component's input field, because all child components and its elements are rendered too:

src/App.test.js

```
describe('SearchForm', () => {
  ...

  it('renders the input field with its value', () => {
    const value = component.root.findByType('input').props.value;

    expect(value).toEqual('React');
  });
});
```

This is our first integration test between the SearchForm and InputWithLabel components, which aren't as tightly coupled as the List and Item components. The InputWithLabel component can be used in other components (highly reusable), whereas the Item component is essentially a non-reusable part of the List component.

src/App.test.js

```
describe('SearchForm', () => {
  const searchFormProps = {
    searchTerm: 'React',
    onSearchInput: jest.fn(),
    onSearchSubmit: jest.fn(),
  };

  ...

  it('changes the input field', () => {
    const pseudoEvent = { target: 'Redux' };

    component.root.findByType('input').props.onChange(pseudoEvent);

    expect(searchFormProps.onSearchInput).toHaveBeenCalledTimes(1);
    expect(searchFormProps.onSearchInput).toHaveBeenCalledWith(
```

```
      pseudoEvent
    );
  });

  it('submits the form', () => {
    const pseudoEvent = {};

    component.root.findByType('form').props.onSubmit(pseudoEvent);

    expect(searchFormProps.onSearchSubmit).toHaveBeenCalledTimes(1);
    expect(searchFormProps.onSearchSubmit).toHaveBeenCalledWith(
      pseudoEvent
    );
  });
});
```

Like the Item component, the last two tests asserted the component's callback handler(s). All input (non function props) and output (callback handler function) props are tested for the SearchForm component's interface and integration with the InputWithLabel component.

You can test edge cases like a disabled button as well. The `update()` method on the rendered test component helps us provide new props to the component at stake:

src/App.test.js

```
describe('SearchForm', () => {
  const searchFormProps = {
    searchTerm: 'React',
    onSearchInput: jest.fn(),
    onSearchSubmit: jest.fn(),
  };

  let component;

  beforeEach(() => {
    component = renderer.create(<SearchForm {...searchFormProps} />);
  });

  ...

  it('disables the button and prevents submit', () => {
    component.update(
      <SearchForm {...searchFormProps} searchTerm="" />
    );

    expect(
      component.root.findByType('button').props.disabled
    ).toBeTruthy();
  });
});
```

Now we'll move one level higher in our application's component hierarchy. The App component fetches the list data, which is provided to the List component. After importing the App component, a naive test would look like this:

src/App.test.js

```
describe('App', () => {
  it('succeeds fetching data with a list', () => {
    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,
      },
    ];

    const component = renderer.create(<App />);

    expect(component.root.findByType(List).props.list).toEqual(list);
  });
});
```

In the actual App component, a third-party library (axios) is used to make a request to a remote API. This API returns data we can't foresee in the test, so we have to mock it instead. Jest provides mechanisms that mock entire libraries and their methods. In this case, we want to mock the `get()` method of axios to return our desired data:

src/App.test.js

```
import React from 'react';
import renderer from 'react-test-renderer';
import axios from 'axios';

jest.mock('axios');

...

describe('App', () => {
  it('succeeds fetching data with a list', () => {
    const list = [ ... ];

    const promise = Promise.resolve({
```

```
      data: {
        hits: list,
      },
    });

    axios.get.mockImplementationOnce(() => promise);

    const component = renderer.create(<App />);

    expect(component.root.findByType(List).props.list).toEqual(list);
  });
});
```

The test reads synchronously, but we still have to deal with the asynchronous data. The component should re-render when its state updates. We can perform this with our utility library and async/await:

src/App.test.js

```
describe('App', () => {
  it('succeeds fetching data with a list', async () => {
    const list = [ ... ];

    const promise = Promise.resolve({
      data: {
        hits: list,
      },
    });

    axios.get.mockImplementationOnce(() => promise);

    let component;

    await renderer.act(async () => {
      component = renderer.create(<App />);
    });

    expect(component.root.findByType(List).props.list).toEqual(list);
  });
});
```

Instead of rendering the App component, we mocked the response from the remote API by mocking the method that fetches the data. To stay on the *happy path*, we told the test to treat the component as an asynchronously updating component. You can apply a similar strategy to the *unhappy path*:

src/App.test.js

```
describe('App', () => {
  it('succeeds fetching data with a list', async () => {
    ...
  });

  it('fails fetching data with a list', async () => {
    const promise = Promise.reject();
```

```
    axios.get.mockImplementationOnce(() => promise);

    let component;

    await renderer.act(async () => {
      component = renderer.create(<App />);
    });

    expect(component.root.findByType('p').props.children).toEqual(
      'Something went wrong ...'
    );
  });
});
```

The data fetching and integration with a remote API is tested now. We moved from focused unit tests for single components to tests with multiple components and their integration with third-parties like axios and remote APIs.

Snapshot Testing

Jest also lets you take a **snapshot** of your rendered component, run it against future captures, and be notified of changes. Changes can then be accepted or denied depending on the desired outcome. This mechanism complements unit and integration tests well, since it only tests the differences of the rendered output without heavy maintenance costs. To see it in action, extend the Item component test suite with your first snapshot test:

src/App.test.js

```
describe('Item', () => {
  ...

  beforeEach(() => {
    component = renderer.create(
      <Item item={item} onRemoveItem={handleRemoveItem} />
    );
  });

  ...

  test('renders snapshot', () => {
    let tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });
});
```

Run your tests again and observe how they succeed or fail. Once we change the output of the render block in Item component in the *src/App.js* file, by

changing the structure of the returned HTML, the snapshot test fails. We can then decide whether to update the snapshot or to investigate the Item component.

Jest stores snapshots in a folder so it can validate the difference against future snapshot tests. Users can share these snapshots across teams for version control (e.g. git). Running a snapshot test for the first time creates the snapshot file in your project's folder. When the test is run again, the snapshot is expected to match the version from the latest test run. This is how we make sure the DOM stays the same.

Exercises:

- Confirm your source code for the last section.
  - Confirm the changes from the last section.
- Add one snapshot test for each of all the other components.
- Read more about testing React components.
  - Read more about Jest and Jest for React for unit, integration and snapshot tests.
- Read more about E2E tests in React.
- While you continue with the learning experience in the upcoming sections, keep your tests green and add new tests whenever you feel the need for it.

React Project Structure

With multiple React components in one file, you might wonder why we didn't put components into different files for the *src/App.js* file from the start. We already have multiple components in the file that can be defined in their own files/folders (also called modules). For learning, it's more practical to keep these components in one place. Once our application grows, we'll consider splitting these components into multiple modules so it scales properly.

Before we restructure our React project, recap JavaScript's import and export statements. 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

```
cd src
touch List.js InputWithLabel.js 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 React from 'react';

const List = ({ list, onRemoveItem }) =>
  list.map(item => (
```

```
    <Item
      key={item.objectID}
      item={item}
      onRemoveItem={onRemoveItem}
    />
  ));

const Item = ({ item, onRemoveItem }) => (
  <div>
    <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>
  </div>
);

export default 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. We'll probably import it eventually.

src/SearchForm.js

```
import React from 'react';

import InputWithLabel from './InputWithLabel';

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

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

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

export default 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 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. Only if a component (e.g. Item) is dedicated to another component (e.g. List) do we keep it in the same file. If a component should be used as a reusable component (e.g. InputWithLabel), it also receives its own 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.

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. Just try to keep the nesting level shallow, otherwise you could get lost deep in folders.

Exercises:

- Confirm your source code for the last section.
  - Confirm the changes from the last section.
- Read more about JavaScript's import and export statements.
- Read more about React Folder Structures.
- Keep the current folder structure if you feel confident. The ongoing sections will omit it, only using the *src/App.js* file.

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 applications. 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 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 section.

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 for its `sortBy` function is encouraged.

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. Headers provide the user more guidance about each column:

src/App.js

```
const List = ({ list, onRemoveItem }) => (
  <div>
    <div 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>
    </div>

    {list.map(item => (
      <Item
        key={item.objectID}
        item={item}
        onRemoveItem={onRemoveItem}
      />
    ))}
  </div>
);
```

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

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 only the List component is in play, so we can lift the state management 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. Further, we added 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>
        <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>
  );
};
```

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

```
import { sortBy } from 'lodash';

...
```

Third, create a JavaScript object (also called dictionary) 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.

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 with `SORTS` at our disposal, we can sort the list before mapping it over each Item component:

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>
  );
};
```

It's done. 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. Again, the initial sort state will be `'NONE'`, meaning it will sort nothing.

Second we rendered more HTML buttons 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 for the last section.
  - Confirm the changes from the last section.
- Read more about Lodash.
- Why did we use numeric properties like `points` and `num_comments` 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.

Reverse Sort

Task: The sort feature works, but the ordering only includes one direction. Implement a reverse sort when the 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.

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, to make it more complex:

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 could 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. 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.

Exercises:

- Confirm your source code for the last section.
  - Confirm the changes from the last section.
- 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 next to each active sort button.

Remember Last Searches

Task: Remember the last five search terms to hit the API, and provide a button to move quickly between searches. When the buttons are clicked, stories for the search term are 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.

First, we will refactor all `url` to `urls` state and all `setUrl` to `setUrls` state updater functions. Instead of initializing the state with a `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 `url` string as state with the state updater function, concat the new `url` 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, concat 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>
  );
};
```

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, removing the last one. 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 twice 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 concats 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, concat the `searchTerm` to the result. If the search terms are identical, return the result without adding anything.

Lastly, the SearchForm'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);
  };

  ...
};
```

Last, 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>
      ...

      <LastSearches
        lastSearches={lastSearches}
        onLastSearch={handleLastSearch}
      />
```

```
      ...
    </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 to follow 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 for the last section.
  - Confirm the changes from the last section.

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 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*.

Task: Rather than fetching only the first page of a list, extend the functionality for fetching succeeding pages. Implement this as an 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.

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 endpoint, 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
```

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 covered 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 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 pending 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 boundaries. Every remote API returns different data structures, so its features may vary, and can be used in applications that consume the API.

Exercises:

- Confirm your source code for the last section.
  - Confirm the changes from the last section.
- Revisit the Hacker News API documentation: 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 an 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.

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.

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 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:

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 for configuring redirects, a 404 page, or headers. Finally, deploy your React application with Firebase in the command line:

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.firebaseapp.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. Try another deployment with `firebase deploy`.

Exercises

- Read more about Firebase Hosting.
- Connect your domain to your Firebase deployed application.
- Optional: If you want to have a managed cloud server, check out DigitalOcean. It's more work, but it allows more control. I host all my websites, web applications, and backend APIs there.

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 or Goodreads would help me provide better content in the future based on your feedback.

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", 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", 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".

- **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, after which you can apply additional tooling.
- **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, especially with React applications. Cypress 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 developer experience. Dive deeper into this topic to make your JavaScript applications more robust. Maybe you 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 component, introducing a UI component library should be easier.
- **Routing:** You can implement routing for your application with react-router. 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 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 to find more interesting topics about web development and software engineering. You can also subscribe to my Newsletter or Twitter page to get updates about articles, books, and courses.

Thank you for reading the Road to React.

Regards,

Robin Wieruch