Array.filter recap

Last updated June 2022

You can **filter** arrays by using the `.filter(callback)` method.
The result will be another array that contains 0 or more elements based on the condition that you have in the callback.

Here's an example:

```
const grades = [10, 2, 21, 35, 50, -10, 0, 1];

// get all grades > 20
const result = grades.filter(grade => grade > 20); // [21, 35,
50];

// get all grades > 30
grades.filter(grade => grade > 30); // [35, 50]
```

Was this helpful?

React intro

Last updated April 2024

React is a JavaScript library created and primarily maintained by Facebook (Meta).

> This course uses React 18. Later this year, React 19 is expected to become stable. Once it becomes stable, the course will be rewritten to conform to the new concepts and best practices in React 19. The new course update will take several months to go live.

Short definition

React is a JavaScript library for building User Interfaces.

Longer definition

React is a JavaScript library that is only responsible for the **view layer** of your application.
This means it is only responsible for rendering your User Interface (such as the text, text boxes, buttons, etc.) as well as **updating** the UI whenever it changes.

For example, say you're building an e-commerce website and would like to maintain the **number of items** in the shopping bag as the user adds and removes items. React makes it easier for you to specify that you'd like to show the number of items in the shopping bag: `{items.length}`.
React will display that (the number of items in the shopping bag) and update it whenever it changes.

It also allows you to reuse this logic in another part of your UI. Say, for example, on the payment page; you can reuse the same logic without having to re-write it.

When learning React (or any other frontend library), it would seem overcomplicated or over-engineered. It is normal to feel that way because the benefit of these libraries is primarily visible when you build medium-sized or large web applications with several team members. So keep that in mind when learning and remember that the goal is to write maintainable and efficient code.

React is a library and not a framework.
The difference between a library and a framework is that a library only helps you in one aspect. In contrast, a framework helps you in many aspects. Let's take an example:

- React is a library because it only takes care of your UI.
- Angular, on the other hand, is a framework because it handles much more than the UI (It handles Dependency Injection, CSS encapsulation, etc.)

React is not about the design of your UI

React itself is not a **UI Library**, as it does not give you beautifully designed buttons or cards.
It helps you manage complicated UI but does not come with a design system.
It will be up to you to choose a design library or use CSS to make it look nice and user-friendly.

What about Web Components?

If you've taken my **Learn JavaScript** course, then you've probably learned about **Web Components**.

You could use Web Components to develop a Web Application; you don't necessarily need React.
You could also use React to complement your Web Components and vice-versa.
The main benefit of Web Components is that they work everywhere. And the main benefit of React is that it moves faster than Web Components. And this is pretty natural because Web Components are a standard (defined and implemented by the browsers), and React is a library that is not standardized.

What are we building in this tutorial?

The final project of this tutorial will teach you how to build the following online supermarket app with *Stripe checkout*. You can try the app on **superm.react-tutorial.app** or play the short video below:

0:00 / 0:51

Recap

- React is a JavaScript library for building User Interfaces.
- React is only responsible for the **View** layer.
- React is not a framework.

Was this helpful?

Install & Setup

Last updated February 2024

To install the **react** package into your project, you need to install it using a package manager (such as **npm** or **yarn**).
We'll use the Node Package Manager (npm) throughout this course.

```
npm install react
```

In this course, all the packages you will need are **already installed**, so you just have to **import** them.

If you'd like to set up a local React project on your computer, then it's recommended that you use **Vite**.
We will be covering this tool in chapter 30. For now, we'll start by explaining concepts at a slow, step-by-step pace to ensure that you understand React entirely.
This makes it easier for you to build more advanced concepts. Trust us—it will work.
We just need your patience and perseverance! 😉

Importing React

React is not part of the browser, so you must **import** it into every JavaScript file where you will need it. Every JavaScript file is a standalone **module**, meaning variables, functions, and imports in one file/module do not affect other files/modules.

Here's how you can import React (assuming you've already installed it):

```
import React from "react";
```

This is called a **default import** because it follows the following syntax: `import Something from "package-name";`.
A named export would be imported like this: `import {Something} from "package-name";`.

Note how we're importing from `"react"`.
This is called a **bare import** because you're not importing from a file path. Example of importing from a file path: `import Something from "./file";`. Importing from a file path always starts with `./`.

It will look for `node_modules/react/` to figure out how to import that package. This is why you need to install react in the first place, which downloads it in your `node_modules` folder.

This import will not work with a `<script type="module">`. In chapter 30, we introduce Vite. Suppose you already want to start working with React and do not want to wait until chapter 30. In that case, we recommend using online editors such as [CodeSandbox](#) and [StackBlitz](#).

The React object

When you import React, you get a React object that contains methods and properties. We will be learning about the methods step by step. Let's start with one of the properties, which is **version**.
A quick side note: A method is a function you need to call with parentheses, whereas a property is a value often pre-calculated.

React exposes its current version through the **version property**; here's how you can read it, assuming you've imported React:

```
console.log(React.version); //"18.1.0"
```

Import Cost

Every time you have an **import** in your code, pay attention that you are adding code to your application.
We'll inform you by telling you the size of every package we mention in this course.
If you've got [VSCode](#) setup on your computer, then make sure to install the [Import Cost extension](#) as it will show you the size of your imports.

The cost of importing React is approximately 6KB.

Recap

- Install react with `npm install react`
- Import React in every file you need it with `import React from "react"`
- Get the current React version with `React.version`

Was this helpful?

Document.createElement

Last updated April 2024

`document.createElement(tagName)` is a Web API provided by browsers that let you programmatically create an HTML element.

Here's a quick refresher on how it works:

```
const element = document.createElement("h2");

// this creates: <h2></h2>
// which you can then insert in the DOM:
document.body.appendChild(element);
```

We're not really interested in inserting this element in the DOM yet, so for now let's focus on this `document.createElement()`.

It creates an HTML element, and we can see all the properties by using `console.dir(element)`, which is a method provided by browsers to list all the properties of a particular object.
Here's what it looks like:

```
▼ h2 i
    accessKey: ""
    align: ""
    assignedSlot: null
  ▶ attributeStyleMap: StylePropertyMap {size: 0}
  ▶ attributes: NamedNodeMap {length: 0}
    autocapitalize: ""
    baseURI: "https://localhost:8082/app.html?id=285"
    childElementCount: 0
  ▶ childNodes: NodeList [text]
  ▶ children: HTMLCollection []
  ▶ classList: DOMTokenList [value: ""]
    className: ""
    clientHeight: 0
    clientLeft: 0
    clientTop: 0
    clientWidth: 0
    contentEditable: "inherit"
  ▶ dataset: DOMStringMap {}
    dir: ""
    draggable: false
    elementTiming: ""
    enterKeyHint: ""
  ▶ firstChild: text
    firstElementChild: null
    hidden: false
    id: ""
```

You won't need most of the properties you see above. However, the most important ones that show up in the full list are:

- id
- style
- className

How would you change the class of the element? What about the style?

```
const element = document.createElement("h2");
element.className = "name-of-class";
element.style = "color: red; background-color: blue";
```

You have to know that this is equivalent of already creating the following element in HTML:

`<h2 class="name-of-class" style="color: red; background-color: blue"></h2>`

The reason why we're creating it in JavaScript will be explained in the following few lessons.

Multiple classes

Multiple classes can be set by separating the class names with a space character.

```
const element = document.createElement("h2");
element.className = "container center";
// <h2 class="container center">></h2>
```

className not class

An important thing to notice here is that changing the element's `class` happens through the `className` property and not `class`.

That's because the keyword `class` in JavaScript is **reserved** and is used for creating a JavaScript class that can be called with `new` (not a CSS class).

You can also change the class with `element.classList.add("name-of-class");` however, you will see later on in React that you will need to avoid changing the DOM directly (for example, with classList.add) and instead define the elements that you'd like to render; more on that later on.

Recap

- `const element = document.createElement(tagName)` creates an HTML element
- `element.style = "background-color: blue"` sets the background-color to blue
- `element.className = "container"` sets the class of the element to: `container`
- multiple classes can be set by separating them with a space character: `element.className = "container center"`

React.createElement

Last updated January 2023

In React, you won't be using `document.createElement`.
However, we saw how it works because it shares **some similarities** with `React.createElement`, but it's not the same thing.

Comparison

Let's compare `document.createElement` and `React.createElement`.

Return value

`document.createElement` returns a **DOM element** (for example a `div` or an `h1`).
Whereas `React.createElement` returns an **object** that represents the DOM element.
The object looks something like this:

```
const element = React.createElement("h1");
//returns an object similar to this one:
{
  type: 'h1',
  props: {}
}
```

The reason why `React.createElement` returns an object rather than the DOM element itself is because React operates a Virtual DOM. We will cover the virtual DOM later on, but for now, here's a simplified explanation of what it means:
A virtual DOM is when a representation of the UI is kept in memory and synced with the DOM.

So `React.createElement()` returns an object rather than a DOM element because this allows React to perform performance optimizations (such as the Virtual DOM).

Changing the class/style

This is where these two methods are similar in terms of syntax:

Notice that we wrote `className` instead of `class` because we're talking about the same properties as the one in the previous lesson.

Writing text

To write a text inside the element, you have to provide the 3rd parameter for React.createElement, called `children` (so it also accepts other Elements later on).

```
React.createElement("h1", {}, "Hello World")
```

This returns an Object representing an `h1` that contains `Hello World`.

Notice how we're saying that we don't want to set a className or style (or other options), so we pass `{}` as the second parameter (you could also pass `null`).

If you think the syntax is a bit ugly and overcomplicated, you're right to think so. We will be using JSX, which replaces the code above with a `<h1>Hello World</h1>`. However, JSX is NOT exactly the same as HTML, so you should learn React.createElement first.

What is a React Element?

In React, an **Element** is the smallest building block.
It represents what the smallest piece of your User Interface will look like. In its simplest example, that could be a paragraph with the text Welcome inside of it (`<p>Welcome</p>`).
You will see later on how we will combine several React Elements as well as advanced logic to build our supermarket shopping Web App.

Recap

- A React Element is the smallest building block.
- It's a representation of a small piece of your UI.
- `React.createElement` returns a React Element
- `React.createElement(type, options, children)`

ReactDOM Intro

Last updated June 2022

Before we get started, try highlighting a line of text and then **save it to your notes**.

A few years ago, React and ReactDOM were part of the same library which was called React. However they got split into 2 separate libraries: React and ReactDOM to allow for the launch of React Native.

React Native is the glue between React and native apps. React Native is outside the scope of this course but as you can see, **React** is the library that lets you write reusable UI and then:

- ReactDOM makes this UI visible in the browser.
- React Native makes this UI visible in a native app.

It's important to remember that the React library has nothing to do with a web browser:

- ReactDOM binds the idea of React to a web browser (example: Firefox, Chrome, Safari, Edge, etc.)

You can find these notes at the top of the page by clicking on the **Note** icon on the top bar. Give it a try!

Popular notes will also be shown in yellow highlights, which can be easily added to your notes with a single click. You can also disable those popular highlights from the top-right menu.

The save notes popup will not appear if you select lines spanning multiple paragraphs. It's because you retain information better when you take shorter notes.

ReactDOM is the glue between React and the DOM.
React creates a virtual representation of your User Interface (in what we call a Virtual DOM). Then ReactDOM is the library that efficiently updates the DOM based on that Virtual DOM.

The Virtual DOM exists to figure out which parts of the UI need to be updated and then batch these changes together. ReactDOM receives those instructions from React and then efficiently updates the DOM.

Why is it a separate library?

the launch of **React Native**.

React Native is the glue between React and native apps. React Native is outside the scope of this course but as you can see, **React** is the library that lets you write reusable UI and then:

- ReactDOM makes this UI visible in the browser.
- React Native makes this UI visible in a native app.

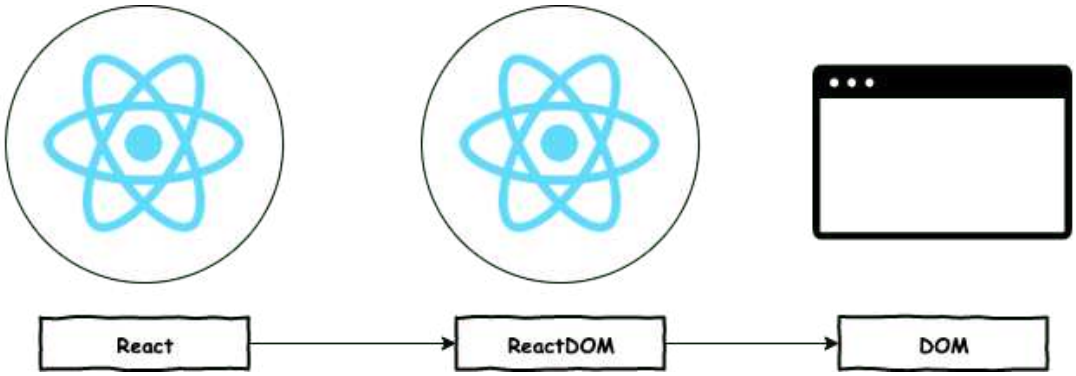
It's important to remember that the React library has nothing to do with a web browser:

- ReactDOM binds the idea of React to a web browser (example: Firefox, Chrome, Safari, Edge, etc.).
- Whereas React Native binds the idea of React to a native app (example: native android, native iOS).

Reconciliation

React is creating the virtual representation of your UI in the memory, and then ReactDOM receives that and syncs your UI (and the changes to it) to the DOM. This process is called **reconciliation**.

As a React developer, this is a feature you get out of the box with React. You do not have to worry about how it works (however, we will see later in this course why immutability is required for reconciliation to work).



Recap

- ReactDOM is the glue between React and the DOM.

- Reconciliation is the process of syncing the Virtual DOM to the actual DOM.

Was this helpful?

ReactDOM Usage

Last updated December 2023

Let's start by installing ReactDOM:

```
npm install react-dom
```

This is something you have to do on top of the previous installation.
So if you were getting started from scratch, you'd have to install both packages (react & react-dom) like this:

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

Import Cost

How much does ReactDOM weigh? ReactDOM weighs 130KB.
So in total, React + ReactDOM weigh 130 + 6 = 136KB.

When developing products for emerging markets or markets with slow internet connections, you may want to consider a lighter alternative such as **Preact**. We will have an introduction to Preact toward the end of this course in an optional chapter (a chapter you can skip).

Importing ReactDOM

We will only need to import one method called `createRoot`. This method is importable from `react-dom/client`:

```
import {createRoot} from "react-dom/client";

// You can call createRoot()
```

> Make sure you import from `react-dom/client` rather than `react-dom`. The latter would fail.

We use ReactDOM to render (visualize) our React Elements on the page.
To do that, you have to tell ReactDOM where to render these elements.
We call that element the **root**.
This is a `<div>` element with an `id` of `root` or `app-root` or `react-root` or whatever name you prefer.
Let's go with `root` for this example:

```
<div id="root"></div>
```

Render your first Element

With the above **root** element, we can now render our first React Element:

```
import React from "react";
import {createRoot} from "react-dom/client";

const root = document.querySelector("#root");
createRoot(root).render(React.createElement("p", {}, "Hello
World"));
```

This will show you a `<p>Hello World</p>` on your screen because the `createRoot().render()` method from ReactDOM receives a React Element (a virtual representation) and then makes it visible in the actual DOM.
Let's break the code down:

1. We start by getting a reference to the root element from the page (using `querySelector` or `getElementById`).
2. We create the root of the React app using `createRoot(root)`.
3. On the result of `createRoot(root)`, we call `.render()` and pass to it our React element.

Does it feel overcomplicated?

You may be wondering if all of this is worth it, but keep in mind that it's totally normal to feel that way because the benefit of React (and other libraries) will only become visible once we start working with larger (and real-world) examples.

There used to be a render method exported from `react-dom`. This got deprecated in React 18 to make way for React 18's concurrent features. If you see the code below, note that it's outdated (but still works in React 18):

```
import React from "react";
// 👎 Outdated code
import {render} from "react-dom";

const root = document.querySelector("#root");
// 👎 Outdated code
render(React.createElement("p", {}, "Hello World"), root);
```

Recap

- Install ReactDOM with `npm install react-dom`
- Import ReactDOM's `createRoot` method with `import {createRoot} from "react-dom/client"`
- ReactDOM weighs 130KB when imported.
- The `root` element is where ReactDOM will render your UI
- `createRoot(root).render(element);`

Was this helpful?

Root Element

Last updated December 2023

We'd like to develop further the concept of the **root** element, which could also be called the **container**.

This **root** element that you pass to `ReactDOM` will become **completely managed by ReactDOM**. So you should not write any kind of JavaScript that changes its content.

So once you have:

```
<div id="react-root">></div>
```

```
import React from "react";
import {createRoot} from "react-dom/client";

const root = document.querySelector("#react-root");
createRoot(root).render(React.createElement("h1", {}, "Hello
World"));
```

You are not supposed to do anything else with the `root` variable and the `<div id="react-root">` ... `</div>` because ReactDOM will manage it.
We say **managed by ReactDOM** because later on, you will be rendering more complicated things than Elements which would update in the future. And ReactDOM will take care of updating these.

Use cases

There are two major use cases for React; here's how they affect the **root** element:

Apps built with React

An app built with React usually has a **single root element**, just like we'll be seeing throughout this course.
The whole application is rendered inside that root element.

Integrate React into an existing App

For example, say you've got a Supermarket website built with Ruby on Rails or Laravel (a backend framework for Ruby and PHP respectively), and you would like to make the **cart** feature interactive, you could decide to use React to make that possible.

In that case, the **cart** logic would go into a `<div id="react-cart"></div>` root element.

And your application could potentially contain more than one root element in the future.

Recap

- ReactDOM completely manages the root element
- You should not directly change/update the content of the root element
- Apps built with React have a single root element (The most common use case)
- Existing Apps that integrate React to make a feature interactive could have more than one root.

Was this helpful?

Chapter recap

Last updated June 2022

You may have noticed that we've been asking you to import React every time you use `React.createElement`; that's because we'd like to make sure you understand this concept and get used to it.

Eventually, this becomes boring, so we'll make it easier for you later in this course.

`React.createElement` is quite a long syntax. However, it's the foundation for JSX. What is JSX, and how does it work? Let's find out in the next chapter.

This is your first chapter recap! You can save it to your notes by clicking on the + icon on the top-right of the recap.

Chapter Recap

- ReactDOM is the glue between React and the DOM.
- ReactDOM is separate from React
- ReactDOM can be installed with `npm install react-dom`
- Import ReactDOM's `createRoot` method with `import {createRoot} from "react-dom/client"`
- `createRoot(root).render(element);`
- Reconciliation is the process of syncing the Virtual DOM to the actual DOM.
- ReactDOM completely manages the root element.
- You should not directly change/update the content of the root element.
- Apps built with React have a single root element (The most common use case).

Was this helpful?

Intro to JSX

Last updated May 2023

Now that you've completed the first three chapters of this course, we recommend that you read the **Features & FAQ** page from the top-right menu.
This will familiarize you with all the features available in this app, as well as the reasoning behind some of them. It also explains the importance of *sleep* when learning something new!

Also, if you'd like to skip to a certain chapter, you can do so from the *Help Center* (bottom-right corner). However, this is discouraged as the course follows a carefully planned structure and many chapters build upon previous concepts.

When working with React, you will need to use React.createElement to represent your UI. However, as you may have seen, the syntax is long. It will become even longer and more tedious when you start having more complicated UI.

React uses a special syntax called **JSX** to solve that issue.
This JSX syntax **may look similar to HTML, but it is NOT HTML**.

Let's see an example:

```
import React from "react";

const title = <h1>Hello World</h1>
```

Even though this looks like it's HTML, it really is not HTML. You will see why in the next lesson.

The code above gets translated to the following:

```
import React from "react";
```

Which one is easier to read?
The first one is because you're expressing that you need to create an element that is an `h1` that contains `Hello World`.

JSX is syntactic sugar for React.createElement

JSX is providing you with syntactic sugar (as in making it nicer to read/write) for the `React.createElement` function.

Instead of having you write `React.createElement` every time, you can write the element in **JSX**.

Always remember that **the JSX that you write is being converted to React.createElement**.
So, JSX is created to make it easier for you to describe your UI.

JSX is NOT part of the browser

Your browser cannot understand JSX because it's a syntax created by React.
You will need a tool (such as **babel**) to convert your JSX code to normal JavaScript (which will contain the React.createElement calls).
We'll discuss that later on with create-react-app and Vite.

This course has been configured in a way that simulates the create-react-app/Vite so you can write JSX, and it will get converted to `React.createElement( ... )`.

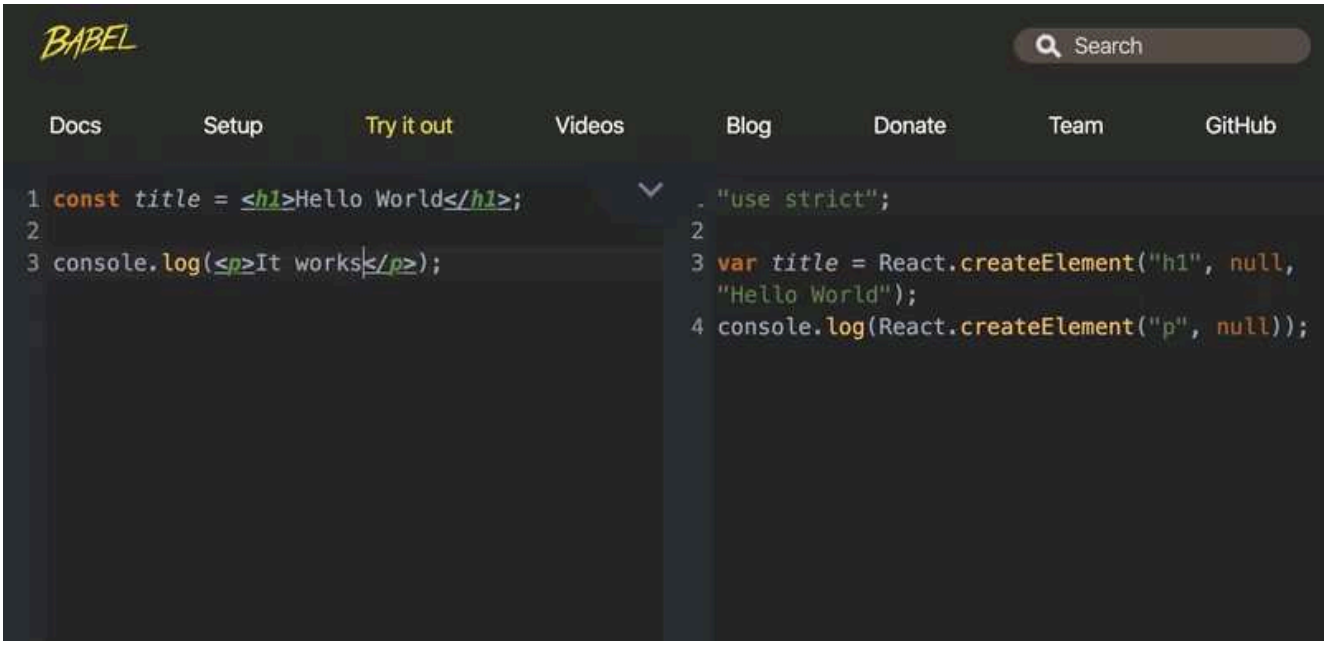
JSX doesn't require React to be in scope (anymore)

Before React 17, you had to import `React` for your JSX to work in every file where you used JSX.

If you're upgrading a source code from React 16, you can leave the `import React;` they won't cause any harm.

Live demo

Here's a short screencast of how the JSX code you write on the left is getting transformed into `React.createElement()` on the right.



You can also give this a try yourself by using the [Babel REPL configured for JSX](#).

JSX with ReactDOM

Here's an example of how JSX is used with ReactDOM:

```
<div id="root"></div>
```

```
import React from "react";
import {createRoot} from "react-dom/client";
```

This will render the `<h1>Hello World</h1>` inside the `#root` element.

Recap

- JSX is a special syntax for React that makes it easier to represent your UI
- JSX looks similar to HTML but it is not HTML
- JSX code you write gets transformed into React.createElement
- JSX is not part of your browser. You need a tool to transform it into valid JavaScript.

Was this helpful?

Attributes

Last updated February 2023

Let's see how we can set attributes to elements in JSX.

```
const title = <h1 id="brand-title">Supermarket</h1>;
```

This is equivalent to the below once converted to JavaScript:

```
const title = React.createElement("h1", {id: "brand-title"},
"Supermarket");
```

What about setting the class?

Remember how we set the class with React.createElement?
Keep it in mind because it's the same in JSX:

```
const title = <h1 id="brand-title" className="primary-
color">Supermarket</h1>;
```

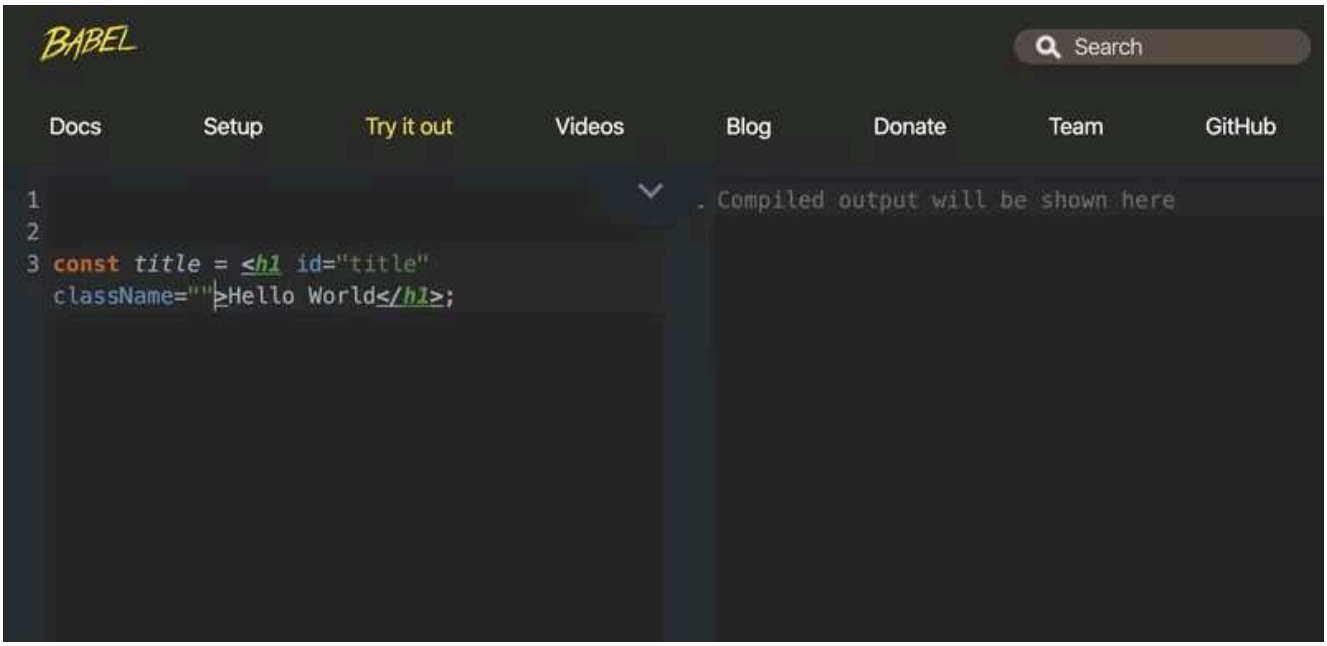
Remember how to set the class; we **have** to use the `className` because it is being set on the element created by React.

This is why JSX is NOT HTML.
With HTML, we would have written: `<h1 id="brand-title" class="primary-color">Supermarket</h1>` but this is not valid JSX.
So don't forget that JSX is converting your code into `React.createElement( ... )`.

Wrap strings with quotes

Live Demo

Just like the previous lesson, here's how your JSX code gets translated to `React.createElement( ... )`



You can also give this a try yourself by using the [Babel REPL configured for JSX](#).

Recap

- Attributes in JSX get passed as the 2nd argument of `React.createElement(...)`
- `<div className="active"></div>` is how you would give the class active to this element.
- You need quotes around attribute values that are strings.

Was this helpful?

Working with JSX

Last updated June 2022

Since JSX is being converted into a `React.createElement( ... )` that returns an object, you can essentially think of a JSX element as if it were an object.
So you can think of `<h1 className="title">Supermarket</h1>` as being an object that more or less looks like this (simplified):

```
{
  type: 'h1',
  props: {
    className: "title",
    children: "Supermarket"
  }
}
```

As a reminder, this object is a representation of a piece of your UI that React maintains in its Virtual DOM.
This lets it efficiently know when it got updated, and then it re-renders it using ReactDOM.

Since `<h1 className="title">Supermarket</h1>` is an object, then you can treat it as an object.
This is why you can:

1. Assign it to a variable:

```
const title = <h1 className="title">Supermarket</h1>;
```

2. Return it from a function

```
function getTitle() {
```

3. Conditionally return different elements:

```
function getTitle(is_open) {
    if (is_open) {
        return <h1 className="title">Supermarket</h1>
    } else {
        return <h1 className="title">Supermarket (closed)</h1>
    }
}
```

4. Other things you can normally do. It's a call to `React.createElement( ... )`.

This may be obvious to some people, but we thought we'd spend some time on it to make sure you understand it because it takes some time to get used to the JSX syntax.

To help you further visualise this, the last example you saw above (`getTitle`), gets transformed to the following code:

```
function getTitle(is_open) {
  if (is_open) {
    return React.createElement("h1", {
      className: "title"
    }, "Supermarket");
  } else {
    return React.createElement("h1", {
      className: "title"
    }, "Supermarket (closed)");
  }
};
```

You may realize that there's a lot of repetition between the code inside the `if` block and the code inside the `else` block.
Don't worry; we will get to that in the following chapter!

Recap

- A JSX element is an object.
- you can treat a JSX element like an object.
- you can assign a JSX element to a variable.
- you can return a JSX element from a function.

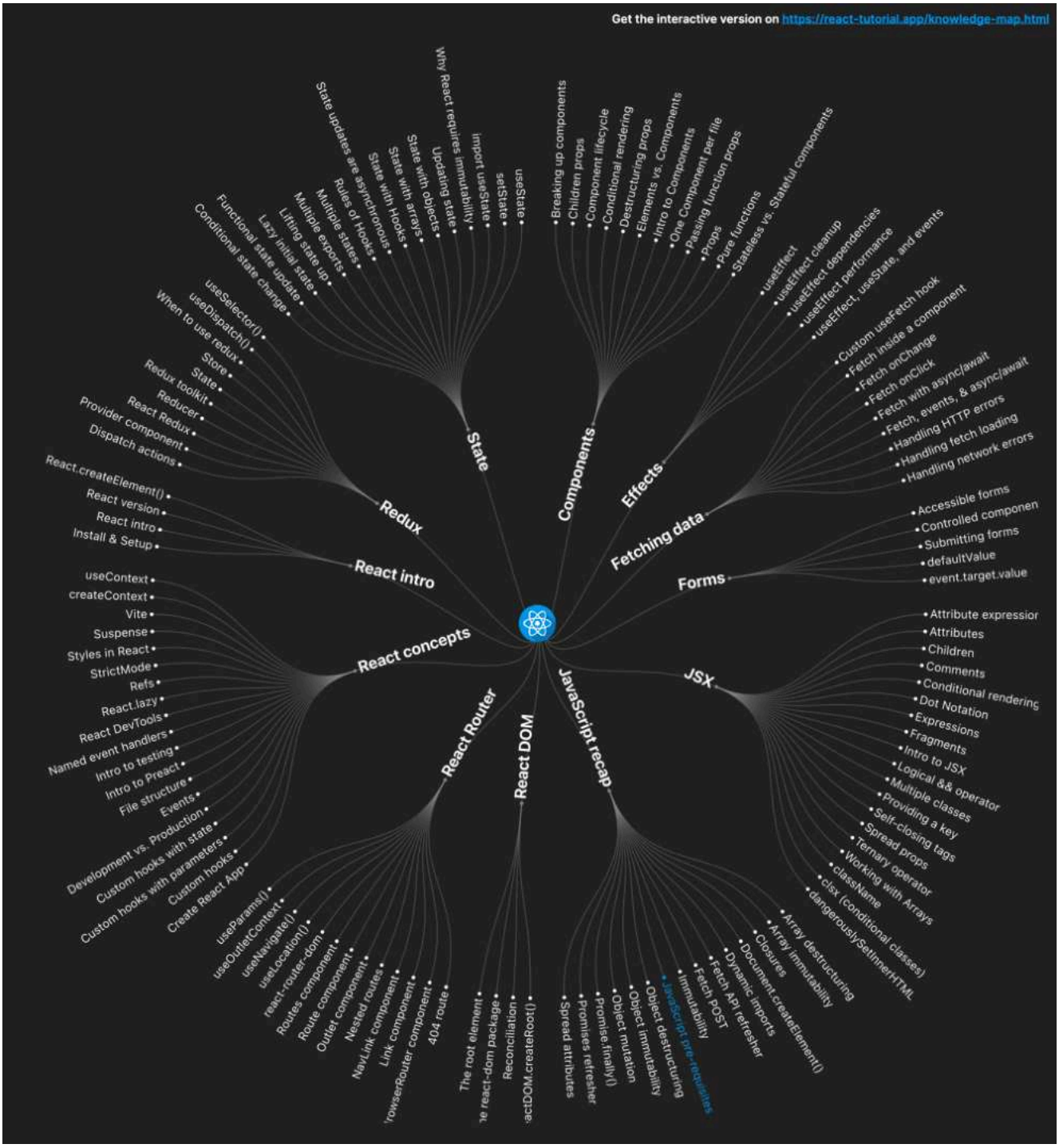
Was this helpful?

JSX Expressions

Last updated September 2023

You've just unlocked your React Knowledge Map ⚡.

This Interactive graph gives you a glance at the topics you've learned and have yet to learn. Completed topics will link back to their relevant lesson.



So far, we've been working with plain React Elements. They always had a specific text and class names that were not dynamic.

It's time to learn about JSX Expressions.

What is an expression?

An expression is **any valid JavaScript code that resolves to a value**.
This means it's any JavaScript code that has an end result, for example:

- `3 + 4`
- `"Sam"`
- `new Date()`
- `2 * 4`
- `name` (assuming the variable `name` has been declared).

And the list goes on.

Every one of these expressions resolves to a value, for example:

- `3 + 4` resolve to the number `7`.
- `"Sam"` resolves to string `"Sam"`.
- `new Date()` resolves to a date object.
- `2 * 4` resolves to the number `8`.
- `name` resolves to the value of that variable, which will most likely be a string.

Expressions in JSX

You can use these expressions in JSX by wrapping them with curly brackets `{}`.

Basic example

This will create an h1 element with the text: `You have 5 notifications`.

See how the expression (2 + 3) gets executed and resolves to 5, which then gets replaced so that the final text is `You have 5 notifications`.
For it to work, expressions **must be inside those curly braces**.

Variables

It's often useful to have variables in those expressions, for example, to show the user's name in the navbar:

```
const user = {
    id: 1,
    name: "Sam"
};

const element = <p className="user-info">Welcome {user.name}!</p>
```

This will create a p element with the text: `Welcome Sam!`.

Function calls

You can also call functions inside those expressions, for example:

```
function capitalise(word) {
    return word[0].toUpperCase() +
word.substring(1).toLowerCase();
}
const name = "brendan";

const element = <p className="user-info">Welcome
{capitalise(name)}</p>
```

Any other JavaScript expression

These were some examples; you can have any kind of valid expression inside the curly braces.

Recap

- An expression is any valid JavaScript code that resolves to a value
- Expressions in JSX need to be wrapped inside curly braces: `{expression}`
- `<h1 className="title">You have {2 + 3} notifications</h1>` is an example of JSX with an expression.

Was this helpful?

Attribute expressions

Last updated June 2022

JSX also supports attribute expressions which means the value of the attribute will be based on an expression (or most often a variable), for example:

```
const limit = 5;

const element = <input type="number" max={limit} />;
```

When converted to `React.createElement()`, it will look like this:

```
const limit = 5;
const element = React.createElement("input", {
  type: "number",
  max: limit
});
```

This example shows how properties can have string values as well as dynamic values; for example, the `max` property takes the value of the variable `limit`.

Quotes OR Expression

It's important to note that you should **either** put **quotes** for the attribute value **OR** use curly braces to have an **expression** with curly braces, but not both.

So this is invalid: `<input max="{limit}" />`, it **will NOT work**.

Instead, it should have been: `<input max={limit} />`

The rule of thumb is:

- If the value is a string, then wrap it with quotes.

We will see more complicated examples in the next lesson, where you'd like to mix both strings and dynamic values.

Numbers and booleans

Number and boolean attribute values should be passed as an expression:

```
<input max={2} disabled={true} className="textbox" />
```

Recap

- If the value of the attribute is a string, then wrap it with quotes
- If the value of the attribute is an expression, then wrap it with curly braces `{}`
- Number and boolean attribute values should be passed as an expression

Was this helpful?

Attribute expressions II

Last updated June 2022

When one of your attributes is partially dynamic, you will have to treat it as if it was entirely dynamic. For example:

```
<li id="item-1"></li>
<li id="item-2"></li>
<li id="item-3"></li>
```

Say you needed to generate these list items with JSX, the `id` is `item-` and then followed by a number.
The number is dynamic, whereas the `item-` is the same.
However, you have to treat it as being entirely dynamic, meaning the attribute's value will have to be wrapped with curly braces `{}`.

There are multiple ways to write it with JSX (assuming a variable `count`):

1. Using string concatenation:

```
<li id={"item-" + count}></li>
```

2. or even better, using template strings:

```
<li id={`item-${count}`}></li>
```

You can use whichever you prefer but make sure you wrap the attribute with curly braces `{}`.

Multiple classes

Another common use case is when working with multiple class names, for example:

```
const active = "active";

const button = <button className={clickable + " " + active}>Click me</button>;
```

Which can also be written using template strings:

```
const button = <button className={`${clickable} ${active}`}>Click me</button>;
```

Later in this course, we will see **clsx**, which is a **tiny library (< 0.5kb)** that makes it easier to deal with multiple conditional classes.

Recap

- When an attribute is partially dynamic, then treat it as being entirely dynamic
- You can use either template strings (with interpolation) or string concatenation.

Was this helpful?

JSX Children

Last updated February 2023

So far, we've only worked with JSX elements that had no children, for example, a `div` or an `h1`.

But your UI will contain more advanced elements, such as the following HTML:

```
<ul>
    <li>First item</li>
    <li>Second item</li>
    <li>Third item</li>
</ul>
```

In JSX, this can be represented like this:

```
const list = <ul>
    <li>First item</li>
    <li>Second item</li>
    <li>Third item</li>
</ul>;
```

The equivalent in `React.createElement( ... )` becomes more complicated and harder to read:

```
const list = React.createElement("ul", null,
    React.createElement("li", null, "First item"),
    React.createElement("li", null, "Second item"),
    React.createElement("li", null, "Third item")
);
```

It works because `React.createElement` takes **all** the arguments passed after the 2nd one and considers them children.

So in this case, the **list** is a `React.createElement("ul", null, child1, child2,`

where `child1` is the first item, `child2` is the second item, etc.

You don't have to write the `React.createElement( ... )` code yourself, but it's good to know how it works.

Gotchas

There are some gotchas and tricky scenarios that we will be discussing in this chapter.

The first one being that later on, you may use **Prettier** to format your code, or you may format your code by yourself, and once you start having more advanced UIs, you will most likely write the **return** on one line, and the rest on another line, like this:

```
function getList() {
    return
        <ul>
            <li>First Item</li>
            <li>Second Item</li>
        </ul>;
}
```

But this **will break** your code because of a JavaScript concept called **Automatic Semi-colon Insertion** (ASI).

What ends up happening here, is that JavaScript will automatically insert a **semi-colon** after your **return** when it gets transformed from JSX to JavaScript, so your code becomes like this:

```
function getList() {
    return;
    React.createElement("ul", null,
        React.createElement("li", null, "First Item"),
        React.createElement("li", null, "Second Item")
    );
}
```

never reaches the `ReactCreateElement("ul", ...)`. This is the correct behavior of JavaScript as it requires compilers to automatically insert a semi-colon (if it doesn't already exist) after certain keywords (more details on **MDN**).

To avoid this issue, always `return ( )` and write the JSX elements inside the parentheses. So the above code becomes:

```
const getList = () => {
    return (
        <ul>
            <li>First Item</li>
            <li>Second Item</li>
        </ul>
    );
}
```

By opening the ( ), no semi-colon will be automatically inserted after `return` until the closing parenthesis is found, and you avoid unexpected issues.

Feel free to try this on **babel**. You will need to wrap the JSX element with ( ) for it to work.

Recap

- JSX supports children
- Always wrap your JSX elements with ( ) when written after a **return**.

Was this helpful?

Self-closing tags

Last updated June 2022

HTML has some self-closing tags, which are HTML tags that cannot contain children; thus, they do not need a closing tag.

An example of a self-closing tag is the `img` because an `<img />` cannot contain children.

Here's a list of the most common self-closing tags in HTML:

- img
- br (line break)
- hr (horizontal rule)
- input

In HTML, you can either write these elements as self-closing or as a normal element, for example:

```
<br>
<br />
```

Both variations are valid in HTML5.

In JSX, you cannot open an element without closing it, thus it's recommended that you use the self-closing tag syntax:

```
const image = <img src="image.png" />
```

This will be especially useful later on once we work with Components.

Recap

Was this helpful?

JSX Fragments

Last updated June 2022

This lesson is the last preparation before we start learning about Components!

When returning elements in JSX, you can **only return one element** at a time.
That element can have children, but you must ensure that you're only returning one element at a time, or else you will get a syntax error.

That's because every Element is an object, and you cannot return two or more objects next to each other.

React.Fragment

React solves this by making you return a Fragment that wraps the elements that you want to return.
So if you want to return the below HTML from a function:

```
<h1>Grocery delivered to your door</h1>
<h2>Free delivery</h2>
<p>Get started now!</p>
```

You will have to use a Fragment that wraps these 3 elements:

```
function getHeroBanner() {
    return (
        <>
            <h1>Grocery delivered to your door</h1>
            <h2>Free delivery</h2>
            <p>Get started now!</p>
        </>
    );
}
```

The fragment is an internal representation in React that lets you wrap multiple elements.

You can choose to return a `<div>`; however, the `<div>` will end up being rendered in your final HTML, whereas fragment tags will disappear (the content of the fragment will be rendered).

Let's say you've got approximately 100 React components. In that case, wrapping each component with an unnecessary `<div>` would make things slower as the size of the DOM would increase.

So we do recommend using the `<>` whenever needed.

React.Fragment (original syntax)

The short syntax for React.Fragment (`<>` and `</>`) was recently introduced.

You might see code using the original syntax with `React.Fragment`:

```
function getHeroBanner() {
    return (
        <React.Fragment>
            <h1>Grocery delivered to your door</h1>
            <h2>Free delivery</h2>
            <p>Get started now!</p>
        </React.Fragment>
    );
}
```

They are essentially the same thing.

Recap

- You can only return 1 element in JSX.
- Wrap multiple elements with `React.Fragment`.
- The short syntax is `<>` that can be closed with `</>`.

Was this helpful?

Intro to Components

Last updated July 2022

From now on, most of the JSX you will write will be encapsulated (contained) inside a **Component**.

What is a Component?

Short definition

A **React Component** is a function that returns **one React Element**, which describes how a section of the User Interface should look.

Longer definition

A **React Component** is a function that returns one React Element (result of React.createElement). That returned Element can have many children as we've seen in the previous chapter.
The **Component** allows you to split your UI into independent, reusable pieces.
This lets you think about each piece of your UI in isolation, making it easier to debug.
A **React Component** is a template or a blueprint that lets you describe a section of the UI; for example, `<Footer></Footer>` is a React Component that describes the **footer**.

Components promote code reuse.

Example

Here's an example of a **React Component** definition:

```
function Footer() {
    return (
```

```
            <p>All rights reserved</p>
        </div>
    );
}
```

Once defined, this **Footer** Component can be used in the same JavaScript file with JSX:

```
import {createRoot} from "react-dom/client";

function Footer() {
    return (
        <div>
            <h3>Company name</h3>
            <p>All rights reserved</p>
        </div>
    );
}

const root = document.querySelector("#root");

createRoot(root).render(<Footer></Footer>);
```

We call this a **function component** because the Component is defined as a **function**.

This course covers function components with Hooks. There's only one optional chapter on Classes towards the end of the course. The reasoning behind this choice is that you can do everything with hooks, and it's easier for you to learn. It becomes easier if you need to learn classes afterward because you know function components with hooks.

UpperCamelCase Component Name

You may have noticed that we called the function **Footer** rather than **footer**. This is necessary. The first character must be in upper case for it to be considered a **Component**; we'll explain why in the next lesson.

- Footer
- ChatMessage
- Button
- ListItem

Recap

- A React Component is a function that returns one React Element describing a section of the UI.
- Components defined with a function are called **function components**.
- React Components promote code reuse and are easier to debug.
- A React Component's name has to start with an uppercase.
- Use UpperCamelCase when naming React Components

Was this helpful?

Components under the hood

Last updated October 2023

In the previous lesson, we used JSX to render the `<Hero></Hero>` Component to the DOM.

Here's how React knows that this is a **Component** rather than an **Element**:

1. React checks the first character of this Component, which is `H` (from `Hero`).
2. Is it an uppercase letter? Then it's a Component.
3. Is it a lowercase letter? Then it's an Element.

From the React documentation: The first part of a JSX tag determines the type of the React element. Capitalized types indicate that the JSX tag is referring to a React component.
Because React treats components starting with lowercase letters as DOM tags, `<hero>` would NOT work in this case; it will render an HTML `<hero>` element rather (even if it's not a valid HTML element) than your own Component `Hero`.

Element vs. Component?

So what's the difference between a React Element and a React Component?

We will ask ourselves this question in a later chapter as we learn more concepts, but for now:
A React Component is a function that returns a React Element.

Composing Components

Here's a glimpse of how your application will look in the future:

```
<App>
    <Navbar>
```

```
        <Hero></Hero>
    </Navbar>
    <BrowserRouter>
        {/* ... */}
    </BrowserRouter>
    <Footer>
    </Footer>
</App>
```

As you can see, it's a mix of Components.
Do you suspect there's a bug in the Navbar? Then you only have to look in the **Navbar** Component.

Self-closing tags

Just like with Elements, you can use the self-closing syntax for Components that do not have any children. So instead of writing `<Logo></Logo>` you can write: `<Logo />`.

JSX to React.createElement

As usual, when you write JSX, it gets converted to React.createElement calls.
Let's see what that looks like for **Components**:

```
const element = <Hero />;
```

gets converted to:

```
const element = React.createElement(Hero, {});
```

As you can see, this is passing a reference to the function `Hero`, which `React.createElement` will then call.
Notice how we're only passing the name of the function `Hero` (without parentheses)

Beware of infinite loops

It's easy to create infinite loops when you're getting started with React; for example, this code will make an infinite loop that will crash your browser - but not React Tutorial ;) Most of the time, we'll be able to notify you if you're about to create an infinite loop.

```
function Button() {
    // Be careful: this is an infinite loop
    return <Button></Button>;
}
```

That's because the `<Button></Button>` JSX will be converted to `React.createElement(Button)` calls which will eventually call the `Button` function. So you'd end up with a function Button that keeps calling itself repeatedly.

In this case, we probably meant to return `<button></button>` (lower case) which is the HTML Element `button` rather than the component `Button`.

Recap

- The first part of a JSX tag determines the type of the React element.
- Capitalized types indicate that the JSX tag refers to a React component.
- A React Component is a function that returns a React Element.

Was this helpful?

One Component per file

Last updated June 2022

An app built with React will have **multiple components**.

A best practice is to define **one component per file** so that you can find it later on (there are exceptions to this, but don't worry about them for now).
It's recommended that the name of the file matches the name of the Component, for example:

- file: `Footer.js` for the Component `Footer`
- file: `AppNavbar.js` for the Component `AppNavbar`

index.js

Your app will still have an `index.js` which is the main entry point of your app (this is sometimes called `app.js`).

So, you define components in other files and then use them in the **index.js**.

Example

Let's see how we can do that by having two files: `Footer.js` and `index.js`:

The `Footer.js` file defines the **Footer** component:

```
//Footer.js
export default function Footer() {
    return (
        <>
            <h3>Footer</h3>
            <p>All rights reserved</p>
        </>
```

Notice how this file will **export default** the **Footer** component.

This is required for us to be able to use the **Footer** component in other files.

```
//index.js
import {createRoot} from "react-dom/client";
import Footer from "./Footer.js";

function App() {
    return (<>
        <Footer />
        <Footer />
    </>);
}

const root = document.querySelector("#root");

createRoot(root).render(<App />);
```

Notice how the `index.js` has to **import** the **Footer** component from `./Footer.js` to be able to use it.

That's because: `<Footer />` gets converted into: `React.createElement(Footer, {})` so for it to work, the function **Footer** has to be in scope, which means it has to be imported.

Also, notice how we rendered the **Footer** twice by just using the component twice in the App component.

Always name your exports

While it is possible to export a component function without giving it a name (an anonymous function), we recommend that you **always** name your exports. This will help you debug your code as you will be able to see the component name when something breaks inside of it, rather than seeing `error in anonymous function`.

```
// This is NOT recommended
export default function () {
    return (
        <h3>Footer</h3>
    );
}
```

Recap

- Define **one component per file** for easier maintenance.
- Make the name of the file match the name of the Component.
- To be able to use a Component that you defined, you need to export it.
- To be able to use a Component defined in another file, you need to import it.

Was this helpful?

Chapter recap

Last updated March 2024

Nice work!

About the instructor

A late introduction: Hey 👋, I'm Jad. I'm the writer of this course, creator, and maintainer of the platform where you're studying.

If you're interested in my frontend tweets, follow me on Twitter **@JoubranJad**.

If you're a fan of cats, you will enjoy my Instagram! **@Jad.of.all.trades**

Please note that I am happy to offer support through this website (Help Center at the bottom-right corner of the page), but unfortunately, not through social platforms such as Twitter/Instagram.

In the following chapter, we'll learn about `Props`. See you there!

Chapter recap

- A React Component is a function that returns one React Element describing a section of the UI.
- Components defined with a function are called **function components**.
- A React Component's name has to start with an uppercase.

- Define one component per file for easier maintenance.
- Make the name of the file match the name of the Component.
- To be able to use a Component that you defined, you need to export it.
- To be able to use a Component defined in another file, you need to import it.

Was this helpful?

Props

Last updated May 2023

Say we've got a component called `<GreetUser>` that says: `Welcome Sam` or `Welcome Alex`.

As you can see, we need to make it say `Welcome` and then the user's name.

Let's start by writing a basic hard-coded version of this component:

```
//GreetUser.js
export default function GreetUser() {
    return <div>Welcome USER</div>;
}
```

This component `<GreetUser />` now displays `Welcome USER`.

Props to the rescue

Instead of rendering `<GreetUser />`, we can render `<GreetUser user="Sam" />`.

This `user="Sam"` is an attribute `user` with a value `Sam` that we've added to our GreetUser component.

Now we can read this `user="Sam"` as an object: `{user: "Sam"}`.

We call that object **props** (which is short for properties).

```
//GreetUser.js
export default function GreetUser(props) {
    console.log(props); // {user: "Sam"}
    return <div>Welcome USER</div>;
}
```

function receives as a first argument.

So we can use an expression to render the user's name (which we can read as `props.user`):

```
//GreetUser.js
export default function GreetUser(props) {
    return <div>Welcome {props.user}</div>;
}
```

- `<GreetUser user="Sam" />` would render `<div>Welcome Sam</div>`
- `<GreetUser user="Alex" />` would render `<div>Welcome Alex</div>`

This makes our component more reusable!

React.createElement

The following JSX code:

```
<GreetUser user="Sam" id="2" />
```

gets transformed into:

```
React.createElement(GreetUser, {user: "Sam", id: "2"});
```

Notice how the JSX attributes on Components get converted into props as the second argument of `React.createElement`.

Recap

- Props is short for properties.
- Props is an object received as the first argument of a Component.
- Attributes on Components get converted into an object called Props.

Was this helpful?

Children props

Last updated June 2022

We've seen so far that `props` represent attributes on a Component.

There's a special type of props for **children**. Let's take an example:

```
const element = <HeroTitle>Welcome!</HeroTitle>
```

The content that goes between the `<HeroTitle>` and `</HeroTitle>` tags is called the **children** props.

You can access it with `props.children`, for example:

```
function HeroTitle(props) {
    return <h1 className="hero">{props.children}</h1>;
}
```

This example of `props.children` is a string (`"Welcome!"`), but in fact, it could be anything else as well.

Not just text

`props.children` could refer to other React Elements or even Components (also multiple Components); here's an example:

```
function Navbar(props){
    return <div className="navbar">{props.children}</div>;
}

const element = <Navbar>
    <HeroTitle>Welcome!</HeroTitle>
    <div>Some content</div>
```

In this example, `props.children` would be an array containing these 3 items:

- `<HeroTitle>Welcome!</HeroTitle>`
- `<div>Some content</div>`
- `<p>Another content</p>`

Even though `props.children` is an array, the end result of the `element` is:

```
<div class="navbar">
    <h1 class="hero">Welcome!</h1>
    <div>Some content</div>
    <p>Another content</p>
</div>
```

As you can see, the navbar is wrapping all the children it receives.
Keep an eye on this pattern, as we'll see it later with the UI Kit and React router.

Recap

- `props.children` represents the content between the tags of a Component.
- `props.children` can be an array or a single item.
- `props.children` can contain text and/or React Elements and/or React Components.

Was this helpful?

Destructuring

Last updated June 2022

Let's start with a recap of **destructuring**, a syntax feature in the JavaScript language.

Destructuring recap

Let's say we've got this object **person**:

```
const person = {
    firstName: "Sam",
    lastName: "Doe",
    age: 24
}
```

and you'd like to create 2 variables, one for the **firstName** and the other one for the **lastName**:

```
const firstName = person.firstName;
const lastName = person.lastName;
```

You could do all of that in one line using the destructuring syntax below:

```
const {firstName, lastName} = person;
```

You could also provide a default value for a variable, in case it did not exist in the object. Example:

```
const {firstName, lastName, status = 'single'} = person;
```

object.

Destructuring props

You don't necessarily have to destructure props, but you will see many tutorials and many developers using it. Thus it's important to be comfortable with this technique.

Since props are an object, you can use destructuring for props.

Before destructuring:

```
function WelcomeUser(props) {
    const username = props.username;
    const notifications = props.notifications;

    return <div>Welcome {username}! You've got {notifications}
unread notifications.</div>;
}
```

After destructuring:

```
function WelcomeUser(props) {
    const {username, notifications} = props;

    return <div>Welcome {username}! You've got {notifications}
unread notifications.</div>;
}
```

Destructuring in the argument

You can also destructure the props variable in the argument, this may be harder to read at first but you will also see it being used in the wild:

```
function WelcomeUser({username, notifications}) {
```

```
}
```

Instead of writing `WelcomeUser(props)` you immediately replace `props` with `{username, notifications}` which is extracting `props.username` and `props.notifications` and creating the 2 variables: `username` and `notifications`.

We don't recommend you overuse this technique as it could make your code hard to read when the number of variables increases.

Recap

- Props are an object, so you can destructure them.
- You can destructure the props in the argument.

Was this helpful?

Chapter recap

Last updated June 2022

Essentially, the difference between **Element** and **Component** is that: Conceptually, components are like JavaScript functions. They accept arbitrary inputs (called "props") and return React elements describing what should appear on the screen.

Another concept we'll learn about is that a component's props should never be modified inside the function.
We will have an entire chapter about it, but first, let's recap this chapter and work on our **first project**!

Chapter recap

- Props is short for properties
- Props is an object received as the first argument of a Component
- Attributes on Components get converted into an object called Props
- `props.children` represents the content between the tags of a Component.
- `props.children` can be an array or a single item.
- `props.children` can contain text and/or React Elements and/or React Components.
- Props are an object, so you can destructure them.

Was this helpful?

UI Kit

Last updated February 2023

Great job on making it this far! To consolidate your knowledge, we're going to have our first project.

Projects aim to help you use what you learned in a more real-life environment.
If you remember, in the first lesson, we said that we'd be building an Online Supermarket app.

When building a relatively large app, it's important to take a step back and think about it before immediately getting to the code.

To maintain consistency throughout our app, we're going to start by building a UI Library (or a UI Kit).

UI Kit

A UI Kit is a collection of all the design elements that you will use in your app, for example:

- Buttons
- Text boxes
- Cards

When building an app, you will either choose a ready-made UI Kit and customize it or build it yourself.
This ultimately depends on the following:

- Your design skills
- The availability of a designer on your team
- The budget for the project

To build a UI Kit, we will create a **React Component** for the User Interface elements we will need. For example, we'll create a `<Button />` Component for our Buttons.

Benefits of building a UI Kit

Building a UI Kit has several benefits; here are the most common ones:

1. Maintain consistency: all the buttons will look the same (we can also create two types of buttons: primary and secondary)
2. Update UI elements in a single place: do you want to change the padding for the button? You can do that in a single Component.
3. Would you like to replace your custom UI Kit with another one? You only need to update the individual components.

Final notes

We haven't learned everything about JSX yet; thus some tasks may seem repetitive, but don't worry about it for now. The goal is to learn and apply what you've learned so far.
Later in this course, we'll learn about Spreading attributes with JSX and other advanced features.

We like this approach where you stumble upon a problem/inconvenience first before we tell you the optimal solution for it.

Enjoy! 😄

Was this helpful?

Project recap

Last updated June 2022

Great job!

You've built your first UI kit out of React Components.

There are some limitations to this kind of UI Kit, for example:

- What if we need to make the `<Input />` disabled? We did not make it support the `disabled` attribute
- What about other attributes that we didn't think about?
- What about the massive amount of code duplicating the props into attributes?
- How do we create several types of Buttons that differ in terms of design? For example, a Primary one and a Secondary one?

We will have an answer for you in the following few chapters.
The next chapter will be about **Pure Functions** which is an important concept that allows React to perform some powerful features.

Was this helpful?

Pure functions

Last updated July 2024

Thank you for being a PRO user!

We recommend you join our GitHub discussions community for PRO users only, where you can ask questions and get product updates. We also have a list of notable discussions where we recommend resources and conferences to stay up to date.

You can join the GitHub discussion by clicking on the **PRO** icon in the top right icon and then selecting the *GitHub discussions* list item.

Some students prefer to navigate the course at their own pace. If you'd like to unlock all chapters, please get in touch with the support using the Help center, and we'll be happy to assist you.

As we talk more about the Virtual DOM, you will see later in this course how React figures out efficiently which components to update when a certain value changes.

For us to help React in achieving optimal performance, it's important to follow one of the most important rules of React, which is:

React components should never modify their props**.**

This means that you should never change the value of the prop inside the component; let's see an example:

```
function Notifications(props) {
    // ❌ do NOT do this (this is changing the props)
    props.data.count = props.data.count - 1;
    return <h3>You have {props.data.count} unread notifications.
</h3>;
}
```

```
};

const element = <Notifications data={notifications} />;
console.log(notifications); // {count: 2}
```

Notice how using the `<Notifications data={notifications}/>` element had a side effect which is changing the value of its props (`notifications.count`).

Instead, it should have been written like so:

```
function Notifications(props) {
    // ☑ this is okay because we're not changing the props
    const value = props.data.count - 1;
    return <h3>You have {value} unread notifications.</h3>;
}

const notifications = {
    count: 3
};

const element = <Notifications data={notifications} />;
console.log(notifications); // {count: 3}
```

So you should think of props as **read-only**.

Same input, same output

To maintain the *same input, same output* philosophy, your component should not modify any variables outside of React. For example, this is **impure** and should be avoided:

```
let count = 0;

function Counter () {
  // ❌ impure - do NOT use
```

```
  return <p>{count}</p>;
}
```

We'll later learn how to use `state` and `effects` (if necessary) to make our components more dynamic while maintaining purity.

Why does React require that?

React requires that because then your functions are **pure**, which means that given the same input (props), they will always have the same **result** (output).

This allows React to quickly identify which Components need to re-render when a certain prop updates. This happens as part of the Virtual DOM.
This becomes very important when your app contains hundreds if not thousands of components that need to be re-rendered.

There are also other benefits with regards to testing & predictability, as well as the new concurrent features in React 18 which we'll discuss later in the course.
It's much easier to test pure functions, and it helps you prevent unexpected bugs.

Recap

- React Components should never modify their props.
- React expects Components to be pure so that it can efficiently update the DOM only when/where necessary.

Was this helpful?

Chapter recap

Last updated April 2024

This was a short chapter. Most courses don't mention pure functions because almost all the functions you write will be pure without you even putting in the effort.

However, we wanted to dedicate a short chapter to ensure you understand why they're important!

This notion will come in handy once we learn about **state**.

Recap

- React Components should never modify their props.
- React expects Components to be pure so that it can efficiently update the DOM only when/where necessary.

Was this helpful?

JSX Gotchas

Last updated June 2022

Let's start by seeing how you write comments in JSX.
To write a comment in JSX, you have to wrap it in an expression and then use a multi-line comment syntax in JavaScript.

As a reminder, here's the multi-line comment syntax in JavaScript:

```
/* This is a comment */
/* Which can
also be written
on multiple lines */
```

In JSX, you need to wrap it with the curly braces because it has to be inside an expression:

```
function Navbar() {
    return <div>{/*this is a comment - the comment won't
render*/}hi</div>;
}
```

This Component is returning `<div>hi</div>` because the rest is a comment.

Conditional rendering

Since JSX are objects representing UI elements, we can use **if** conditions to make our components a little more powerful.
For example, we can return different React Elements from the Component based on the **props**!

```
function Navbar(props) {
```

```
    }
    return <p>Welcome back!</p>
}

const element1 = <Navbar loggedIn={true} />
const element2 = <Navbar loggedIn={false} />
```

Recap

- Comments in JSX use the following syntax: `{ /* comment */ }`
- Conditional rendering allows you to return different Elements based on props (or other conditions).

Was this helpful?

Conditional classes with clsx

Last updated May 2023

You may have noticed so far that it's a bit of a pain to apply classes dynamically in React because you have to build the `className` string manually.

There are tiny libraries that help with that; the most popular ones are:

- classnames
- clsx

Both libraries are tiny; you can use whichever one you want. In this course, we'll use `clsx` because it's a bit more modern.

Installation

To install **clsx**, run:

```
npm install clsx
```

Import Cost

The `clsx` package is approximately 300 Bytes only (less than 1KB), so you don't have to worry about its size when importing it into your project.

Importing clsx

You can use clsx in React Tutorial in any challenge by importing, as we've already installed it for you.

To import **clsx**, you have to import clsx from the package name:

Conditional classes

Now that we've imported clsx, let's see how it can simplify conditional classes:

```
import clsx from "clsx";

const result = clsx({
    "link": true,
    "link-primary": true
});

console.log(result); //"link link-primary"
```

So we gave clsx an object containing:

- `link` as a key, with `true` as a value
- and `link-primary` as a key, with `true` as a value and it returned a string containing both classes separated by space.

Let's take another example:

```
import clsx from "clsx";

const result = clsx({
    "link": true,
    "link-primary": false
});

console.log(result); //"link"
```

That's essentially the same thing as the previous example. Still, this time we've said that `link-primary` is `false`, so the resulting string did not include `link-primary`.

```
import clsx from "clsx";

function MyComponent(props) {
    const className = clsx({
        "title": props.loggedIn
    });

    return <h1 className={className}></h1>
}

const example1 = <MyComponent loggedIn={true} />; //
className="title"
const example2 = <MyComponent loggedIn={false} />; //
className=""
```

In the code above, `props.loggedIn` is a boolean. We use that boolean value to conditionally add/remove the `title` class:

- when `props.loggedIn` is `true`, clsx receives `{"title": true}` and in that case outputs the string `"title"`.
- On the other hand, when `props.loggedIn` is `false`, clsx receives `{"title": false}` and in that case outputs an empty string `""`.

So `clsx` allows us to have conditional classes without having to write `if` conditions. Here's how the above example would look with `if` conditions and without `clsx`:

```
function MyComponent(props) {
    let className = "";
    if (props.loggedIn) {
        className = "title";
    }

    return <h1 className={className}></h1>;
}
```

```
import clsx from "clsx";

function MyComponent(props) {
    const className = clsx({
        "text": true, // this class is always added
        "title": props.loggedIn
    });

    return <h1 className={className}></h1>
}

const example1 = <MyComponent loggedIn={true} />; //
className="text title"
const example2 = <MyComponent loggedIn={false} />; //
className="text"
```

Notice how the class `text` is *always* added, whereas the class `title` is only conditionally added, depending on the boolean value of `props.loggedIn`.

Recap

- You can install `clsx` with `npm install clsx`
- `clsx` is a tiny utility for constructing className strings conditionally.
- `clsx({"your-class-name": booleanValue})` is the generic syntax for clsx.

Was this helpful?

Spread attributes

Last updated August 2022

Let's start with a recap of JavaScript's **spread operator**:

```
const person = {
    id: 1,
    name: "Sam"
};

const details = {
    age: 23,
    loggedIn: true
};

const person_details = {...person, ...details};
/*
{
    id: 1,
    name: "Sam",
    age: 23,
    loggedIn: true
}
*/
```

Notice how `...person` spreads all the keys and values of the object into the new **person_details** and the same thing for `...details`.

Let's take a look at another use case of **spread operator**:

```
const person = {
    id: 1,
    name: "Sam",
    age: 23,
}
```

```
const {id, ...rest} = person;
console.log(id); //1
console.log(rest); //{name: "Sam", age: 23}
```

Notice how we're destructuring the **id** from `person` (which is equivalent to `person.id`), and then, later on, we ask for **all the other keys and values** to be saved in a new object called **rest**.

And thus, you end up with a new object containing all the other keys and values (without the `id` because it has already been destructured).

In this example, the `...` is called the *rest property* as you're extracting all the *rest* (remaining) properties.

JSX Spread Attributes

Sometimes it's useful to do the same in JSX, for example, you've got a Component that receives many props, and you'd like to take the **Children** out and then destructure all the rest into a new variable and then pass those down to an element:

```
function Navbar(props) {
    const {children, ...rest} = props;
    return <h1 {...rest}>{children}</h1>;
}
```

Notice how we start by destructuring the **children** from props, then we create a new object called **rest** which contains all the other props.
And finally, we **spread** this object into the `<h1>`, which is effectively passing every prop to the `<h1>` element.

So say you'd use the Navbar as: `<Navbar tabIndex="2" className="navbar">Hello World</Navbar>`, then you'd get a variable `children`

```
{
    tabIndex: "2",
    className: "navbar"
}
```

and then your Component will return the following React Element:

```
<h1 tabIndex="2" className="navbar">Hello World</h1>
```

Imagine we add a new attribute, for example: `disabled={true}`, then your code still works and will still apply the `disabled` attribute on the **h1** without having to update our Component.

When to use it?

While spread attributes can be useful, they also make it easy to pass unnecessary props to components that don't care about them.

The most common use case of Spread Attributes is when you're building a UI Kit and want to make it possible to customize the rendered element.

Recap

- Spreading attributes in JSX allows you to take all the props (or a portion of the props) and pass them down to the element being rendered without having to write all the props one by one.
- Spreading attributes is mostly used when building UI Kits.

Was this helpful?

Chapter recap

Last updated February 2024

Chapter recap

- Comments in JSX use the following syntax: `{ /* comment */ }`

- clsx is a tiny utility for constructing className strings conditionally.
- `clsx({"your-class-name": booleanValue})` is the generic syntax for clsx.
- Spreading attributes in JSX allows you to take all the props (or a portion of the props) and pass them down to the element being rendered without having to write all the props one by one.
- Spreading attributes is mostly used when building UI Kits.

Was this helpful?

Improved UI Kit

Last updated June 2022

Congratulations on making it this far!

Before starting the project, here's an extra piece of information that might come in handy.

`clsx` supports multiple arguments. It will return a className string that contains all of them (separated by a space character). For example:

```
import clsx from "clsx";

const classes = clsx("class1", "class2");
console.log(classes); //"class1 class2"
```

This is useful because you can also use variables with it, for example:

```
import clsx from "clsx";

let dynamicClass = "class1";
const classes = clsx(dynamicClass, "class2");
console.log(classes); //"class1 class2"
```

The `dynamicClass` in this example is set to `"class1"`. However, the code can be more advanced. For example, `dynamicClass` might later be re-assigned. `clsx` will return the `className` based on the arguments it receives.

Let's get coding!

In this project, we'll improve our previously built UI Kit by taking advantage of the new advanced concepts we learned.

Was this helpful?

Array destructuring intro

Last updated June 2022

In this chapter, we'll deep dive into array destructuring because that's an important prerequisite for **Hooks** (a React concept).

Even if you already know array destructuring, keep reading as we'll cover some advanced concepts.

Basic Array Destructuring

Let's start with the most basic example of Array Destructuring.

Array destructuring allows you to have a shorter syntax when reading multiple entries of an array. For example, instead of writing:

```
const dimensions = [20, 5]

const width = dimensions[0]; //20
const height = dimensions[1]; //5
```

You can create the two variables `width` and `height` in a shorter syntax, which is called array destructuring:

```
const dimensions = [20, 5]

const [width, height] = dimensions;
```

The way it works is that `width` will be assigned to `dimensions[0]` (because it was the first item in the destructuring), and `height` will be assigned to `dimensions[1]` (because it was the second item in the destructuring).

that will be destructured.

Here's another example of array destructuring:

```
const point = [3.3124, 14.52028, 10]; //coordinates

const [lat, lng, elevation] = point

console.log(lat); //3.3124
console.log(lng); //14.52028
console.log(elevation); //10
```

Remember that this is called syntactic sugar, which makes the code easier to read (you may disagree with that at first, but once you get used to it, it becomes easier to read).

Arrays containing various data types

Array destructuring still works even if the array contains various data types, for example integers, strings, objects, functions, etc.. For example:

```
//data contains the name, followed by the age, followed by the details object
const data = ["Sam", 23, {
    id: 1,
    created_at: "Jan 19"
}];
```

You can destructure this `data` array into three variables using the usual destructuring syntax:

```
const [name, age, details] = data;
```

and you will get:

- the `name` which is `"Sam"`

We're showing you that destructuring works no matter the array's content.

It could even contain a `function` or a `boolean`.

Recap

- Array destructuring is a shorter syntax for creating variables out of the items of an array.
- Array destructuring works regardless of the data types in the array.

Was this helpful?

Array destructuring advanced

Last updated June 2022

Say you have the function **getUser** below:

```
function sayHello() {
    return "Welcome!";
}

function getUser() {
    //return the id and a function that welcomes the user
    return [15, sayHello];
}
```

What is this function returning?

1. You should first notice that `getUser()` returns an array.
2. Then, you should notice that this array contains two items.
3. The first item is a `number`.
4. The second item is a `function`.

`sayHello` is a function reference that we can call later (we'll see how).

But now, question: Can we still use destructuring after we call `getUser()`?

```
const result = getUser();

const id = result[0];
const sayHello = result[1];
```

Can we destructure `getUser()`?

Well, we can! It's technically the same syntax, but we thought we'd break it down because it might be confusing at first.

Let's do it step by step:

```
const [id, sayHello] = result;
```

This is how we destructure the `result` variable, but we can go a step further and destructure `getUser()` directly without having to create an extra variable (`result`):

```
const [id, sayHello] = getUser();
```

So we're destructuring `id` and `sayHello`, which correspond to index 0 and 1, respectively, of the returned data from `getUser()`.

How do we call sayHello?

Since `sayHello` is a function, you can call it by adding `()`. Here's the full example:

```
function sayHello() {
    return "Welcome!";
}

function getUser() {
    //return the id and a function that welcomes the user
    return [15, sayHello];
}

const [id, sayHelloFunction] = getUser();
console.log(id); //15
console.log(sayHelloFunction()); //Welcome!
```

It may be unclear **why** we'd want to do that, but stay tuned for the next chapter!

Recap

- You can destructure from a function, assuming the function is returning an array of elements

Chapter recap

Last updated March 2023

As you can see, array destructuring is just syntactic sugar as it makes it easier for us to create variables out of elements of an array.
You may have also noticed that returning an array from a function lets us return **several** variables (which could also be of different types).

This concept is available in many programming languages. Still, it is also a prerequisite for React Hooks, which we'll be diving into over the next few chapters!

I hope you're excited!

Recap

- Array destructuring is a shorter syntax for creating variables out of the items of an array.
- Array destructuring works regardless of the data types in the array.
- You can destructure from a function, assuming the function is returning an array of elements

Was this helpful?

State with Hooks

Last updated April 2023

In this chapter, we'll learn about **state** in React while using **hooks**.

We will be focusing on the **state** more than **hooks** for now, but don't worry, as we will spend *several chapters* explaining state and hooks step by step, just like you're used to in this course.

What is State?

State refers to any variable defined inside a Component with the intent to update later.

For example: when creating a `<Stopwatch />` component, we need to keep track of the elapsed seconds. The elapsed seconds cannot be a props for 2 reasons:

1. It should be contained inside the `<Stopwatch />` rather than passed from outside
2. It should update/change! Props are supposed to be pure and should not be updated/changed.

If you're wondering what we mean by "it should be contained inside the Component rather than passed from outside", remember that props are passed from outside, for example:

```
<Stopwatch theme="dark" />
```

The **theme** is a prop because it's passed from outside.
The **theme** cannot be changed from within the **Stopwatch** component; it has to change from the outside. Once it's changed from the outside, it gets processed again by the component.

This is why **theme** is a prop. We don't plan on updating it from inside the **Stopwatch**.

What happens when the state changes?

When the state is updated, ReactDOM will efficiently and automatically **re-render** the Component to the DOM. We'll see how this works soon.

Another example

Let's take another example.

Say you're creating an `<Item />` component that contains a counter.
The user can then increment/decrement the counter using the + and – buttons as they decide how many items they would like to order.
The `counter` is a **state** variable that will be updated when the user clicks on the + and – buttons.
When the user clicks these buttons, the `counter` will re-render on the UI automatically.
So `0 item(s)` would show `1 item(s)` when the user clicks on the + button.

We will build this component in the following few chapters (once we learn about State, Hooks & Events), so stay tuned!

Recap

- **State** refers to any variable defined inside a Component with the intent to update later.
- State variables can be updated from inside the component.
- When a state variable is updated, the component automatically re-renders.

Was this helpful?

Import useState

Last updated June 2022

Let's start by creating our first state variable.
To do that we need to import the `useState` function from the `"react"` package.

`useState` is a named export which means the syntax for importing it is:

```
import {useState} from "react";
```

If you were already importing `React` in the same file, you can merge the imports into a single one. For example:

```
import React from "react";
import {useState} from "react";
```

The above can be combined into a single import:

```
import React, {useState} from "react";
```

The syntax may look weird at first, but remember that:

1. `React` is a **default export** (no curly braces)
2. `useState` is a **named export** (wrapped in curly braces)

useState is a React Hook

`useState` is one of many hooks provided by React.
We won't be covering what Hook means for now, but a detailed explanation will follow in the next few chapters.

Recap

- `useState` allows us to create a state variable in a Component
- `useState` is a named export that needs to be imported
- You can import `useState` and `React`: `import React, {useState} from "react";`
- `useState` is one of many React Hooks.

Was this helpful?

useState

Last updated March 2023

Now that we know how to import the `useState` function, let's create our first state variable.

For that, we need to call the `useState` which has the following signature `useState(initial_value)`.

The **initial_value** is the value that the state will have in the beginning.
For example, to create a `<Stopwatch />` Component, we'll need a `seconds` state variable and that state variable will **start at 0**.
Which means that the initial value is `0`.

So here's how you create the state: `useState(0)`.

What does useState return?

This will sound familiar because we spent an entire chapter breaking it down 😄
`useState` returns an array of 2 items:

1. the first one is the **current value of the state**
2. the second one is a function that updates the state (this will be explained later)

So instead of writing:

```
const result = useState(0)
const seconds = result[0];
const setSeconds = result[1];
```

We will use the array destructuring syntax:

```
const [seconds, setSeconds] = useState(0);
```

We will explain later.

It is important to name the destructured properties as follows:

1. The first one should be called the name of the state (in this example `seconds`)
2. The second one should be called `set` and then followed by the capitalized name of the state (in this example `setSeconds`)

This is important as your Components will get more complicated, it's necessary to know that `seconds` is the **state** and `setSeconds` is the function that updates the **seconds** state.

Full example

Let's take a full example using state:

```
import {useState} from "react";

function Stopwatch() {
    const [seconds, setSeconds] = useState(0);

    return <div>{seconds}</div>
}
```

Notice how we call `useState(0)` inside the **Stopwatch** Component.

Important things to note:

1. You should **only** call `useState` **inside** the Component, not outside.
2. The `useState` should be the **first thing** you call inside that function (we'll explain why later on)

Also note that because the `seconds` state is a variable, you can use it as an expression inside JSX: `<div>{seconds}</div>`.

We will not update the value of `seconds` in this chapter, so it's normal that you feel the Component is not much smarter than before.

Also note that when we create the state **items**:

```
const [items, setItems] = useState(0)
```

then **items** will be equal to 0 because that's the initial value that we passed.

Recap

- The `useState()` function takes the initial_value as the only argument.
- The `useState()` function returns an array of 2 items
- Always destructure the useState into `abc` and `setAbc` where `abc` is the name of the state
- The default value of the state variable returned by useState will be the same as the initial_value passed to `useState()`.

Was this helpful?

Chapter recap

Last updated August 2023

Great work!

In this chapter, we only learned how to create and use *state* in JSX.
In the next one, we'll learn how to update and how React automatically re-renders the Component.

Also, it's important to not create *state* variables when they can be computed from a `prop` value.

Here's a bad example that you should **avoid**:

```
import { useState, useEffect } from 'react';

function Products(props) {
    const { productsList } = props;
    // ❌ Unnecessarily complicated
    const [count, setCount] = useState(0);

    useEffect(() => {
        setCount(productsList.length);
    }, [productsList]);

    return <p>Number of products: {count}</p>;
}
```

We don't need to create a state variable because we can directly compute that from the props:

```
import { useState, useEffect } from 'react';

function Products(props) {
    const { productsList } = props;
    // ✅ compute it from the props
```

```
    return <p>Number of products: {count}</p>;
}
```

Recap

- **State** refers to any variable defined inside a Component with the intent to update later on.
- State variables can be updated from inside the Component.
- When a state variable is updated, the Component automatically re-renders.
- `useState` allows us to create a state variable in a Component
- `useState` is a named export that needs to be imported
- You can import useState and React: `import React, {useState} from "react";`
- `useState` is one of many React Hooks.
- The `useState()` function takes the initial_value as the only argument.
- The `useState()` function returns an array of 2 items
- Always destructure the useState into `abc` and `setAbc` where `abc` is the name of the state
- The default value of the state variable returned by useState will be the same as the initial_value passed to `useState()`.

Was this helpful?

Changing state

Last updated June 2022

In this chapter, we'll see how to update the state and how React & ReactDOM will automatically re-render the Component and show us the updates in the browser.

A quick intro to Events

We have several chapters entirely dedicated to **events** in React.
However, we'd like to have a quick intro now about the most basic Event you can achieve in React in order to update the state when the user clicks on a `<button>`.

So in this project, we'll only work with **click** events on the `<button>` element.

To add a click event on a `<button>`, we have to add the `onClick` (case sensitive) attribute, and assign it to an arrow function: `() =>`

```
function Welcome() {
    return <button onClick={() => console.log("button was clicked")}>Click me</button>
}
```

This will log `button was clicked` every time you click on the **button**.

Let's break down the syntax `onClick={() => console.log("button was clicked")}`:

- `onClick={}` JSX attribute with a dynamic expression.
- `onClick` adds an event listener
- `() => console.log("button was clicked")` is an arrow function. Because the body of the arrow function is just 1 line, we can omit the `{}`. You can also write: `() => {console.log("button was clicked")}`

State is a variable that we can update at some point in the future.
For example, when the user clicks a button or when you get data from an API.
For now, we'll stick with a simplified example to isolate the concepts you're going to learn.

In this chapter, we'll update the state when a button is clicked.

To update the state, **you always have to use the 'set' function that you get back from** useState.

So if we created a state called `seconds`, then we have to use the `setSeconds(newValue)` function that we destructured. Let's use the same `Stopwatch` Component as before:

```
import {useState} from "react";

function Stopwatch() {
    // hooks have to be at the top
    const [seconds, setSeconds] = useState(0);

    return (<>
        <h2>{seconds}</h2>
        {/* increment seconds state by 1, when you click on the
button*/}
        <button onClick={() => setSeconds(seconds + 1)}>Click to
add 1</button>
    </>);
}
```

Notice how we update the state by calling **setSeconds()** and passing to it the new value of the state.
The new value of the state is `seconds + 1`, which is effectively: the current value + 1.
That's because `seconds` holds the current value of the state, so `seconds + 1` will increment it by 1.

to worry about it for now. We discuss that in chapter 06 (functional state updates).

Recap

- `onClick={() => console.log("Hello World")}` is a click event that works on a `<button>` and logs "Hello World"
- You always have to use the 'set' function that you get back from useState (for example: setSeconds, setCount)

Was this helpful?

Breaking down useState

Last updated June 2022

Now that we learned about both values returned by `useState` we can break down how it works:

```
import {useState} from "react";

function Stopwatch() {
    const [seconds, setSeconds] = useState(0);
    console.log("value of seconds is: ", seconds);

    return (<>
        <h2>{seconds}</h2>
        <button onClick={() => setSeconds(seconds + 1)}>Click to
add 1</button>
    </>);
}
```

The first time this Component renders, we will get the following console log:

```
value of seconds is: 0 // it uses the default value passed to
useState, which is 0
```

Then when the user clicks on the button, we set the seconds' new value to `seconds + 1`, so `0 + 1` = 1

Because we called `setSeconds`, React knows that this Component's state has changed, which means it needs to re-render, so the whole `Stopwatch` function is called again, but this time the `seconds` state has already been created so it will not be set to 0.

React will NOT use the initial value anymore because the state already exists.

So this time, `const [seconds, setSeconds] = useState(0)` actually gives the state `seconds` which has a value of 1.

So here's the console log that we see when React re-renders this Component:

```
value of seconds is: 1 // that's what we set it to when we
called setSeconds(1)
```

Then when the user clicks on the button, we set the seconds' new value to `seconds + 1`, so `1 + 1` = **2**

Recap

- When you change the state, React will call the Component function again to re-render
- The `initialValue` passed to `useState(initialValue)` is only used the **first time** a Component renders.

Was this helpful?

SetState

Last updated June 2022

Why do we need a dedicated method to update the state?

In short, React will not be able to know if you change the value of `seconds`, so we need to use the method provided which is `setSeconds`.

This method will **hook** into the internals of React and notify it that we are changing the value of a state (hence why useState is a hook, more on that in a dedicated chapter).

This often means that the Component has to re-render, which means it has to be re-drawn on the screen, which is taken care of by ReactDOM.

Recap

- `useState` is a React Hook that *hooks* into the internals of React and notifies it that a state variable has changed.

Was this helpful?

Chapter recap

Last updated July 2022

It's sometimes hard to remember the big picture, so let's take a step back and see *why* we're using React and how it's helping us.

```
import {useState} from "react";
import {createRoot} from "react-dom/client";

function Counter() {
    const [count, setCount] = useState(0);

    return (<>
        <div>{count} times clicked!</div>
        <button onClick={() => setCount(count + 1)}>Add
1</button>
    </>);
}

createRoot(document.querySelector("#root")).render(<Counter />);
```

The benefit of using React here is that we `return <div>{count} times clicked! </div>`, and we do NOT have to worry about the updates.
We call `setCount()` and React will worry about the `{count}` value in `<div>{count} times clicked!</div>`.

Typically with Vanilla JavaScript (JavaScript without any library), you'd have to manually update the DOM every time a variable changes, and that could quickly become messy as the requirements get more advanced.

Chapter recap

- `onClick={() => console.log("Hello World")}` is a click event that works on a `<button>` and logs "Hello World"

- When you change the state, React will call the Component function again to re-render
- The `initialValue` passed to `useState(initialValue)` is only used the **first time** a Component renders.
- `useState` is a React Hook that hooks into the internals of React and notifies it that a state variable has changed.

Was this helpful?

JavaScript Closures

Last updated April 2024

Closures is one of those concepts in JavaScript that appears to be complicated, but in reality, it's not. So what's a closure?

A closure is where an inner function has access to the outer function's variables. Let's take a look at the example below:

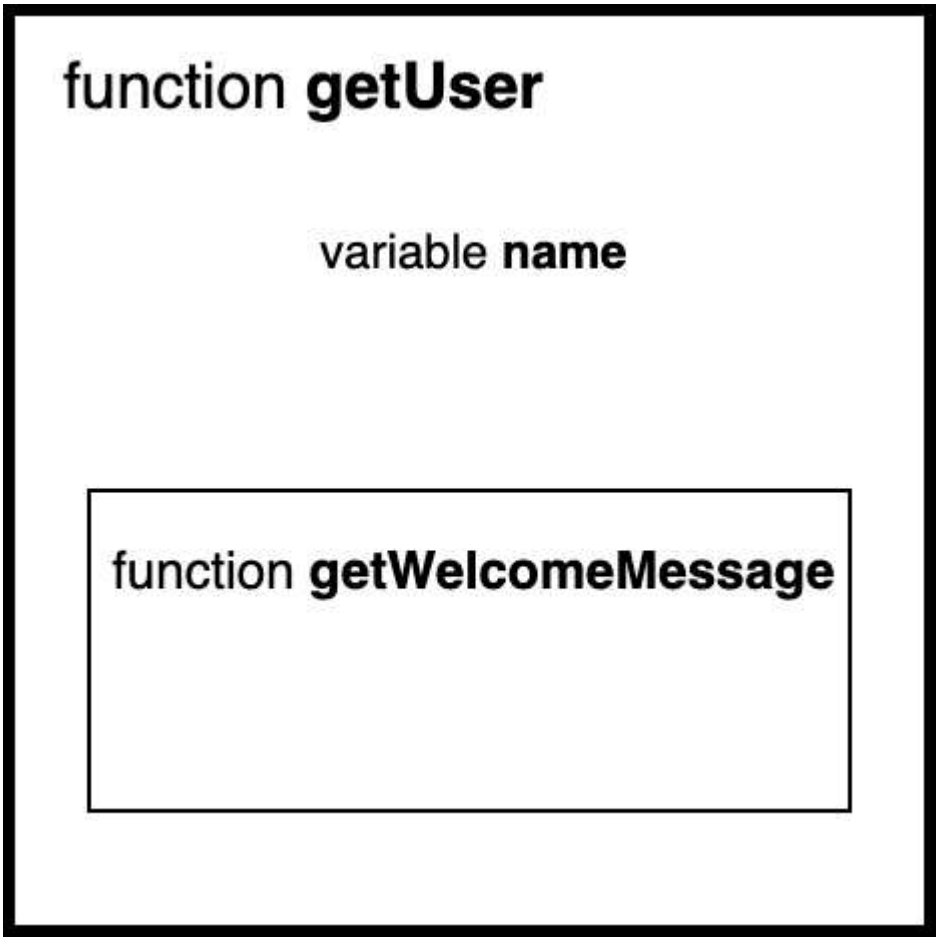
```
function getUser() {
    const name = "Sam";

    function getWelcomeMessage() {
        return `Hello ${name}`;
    }

    return {
        name: name,
        message: getWelcomeMessage()
    }
}
```

Notice how we've got a closure? It's because we've got a function `getWelcomeMessage` inside the `getUser` function.
Here's an explanation of the code:

- The `getUser` function defines a variable `name` which is equal to "Sam".
- Then inside of that function, we define a function `getWelcomeMessage` which returns `Hello ${name}`.
- Then we `return` an object containing the `name` and the `message`
- Where `message` is the result of `getWelcomeMessage()`.

You may have noticed that the variable `name` is available to the `getWelcomeMessage` because `getWelcomeMessage` is defined inside `getUser`.

inside `getUser`, in this example, `getWelcomeMessage`.



Looking at this visual above, look at how `getUser` defines a `name` variable, which then becomes available in any function defined inside `getUser`.

So because `getWelcomeMessage` is defined in `getUser`, you will be able to use the variable `name`.

That's it!

Note that the concept of closures also applies to any function, not necessarily a function within a function, for example:

```
const name = "Sam";

function getUser() {
    return name; // this works because the function has access to
the outer scope.
}
```

We're going to focus on examples you will actually use in real-life projects and in React.

Also applies to arrow functions

The same concept also applies for arrow functions, so if you rewrite it with arrow functions:

```
const getUser = () => {
    const name = "Sam";

    const getWelcomeMessage = () => {
        return `Hello ${name}`;
    }

    return {
        name: name,
        message: getWelcomeMessage()
    }
}
```

The same concept still applies, and we will get the same result.

Why is it useful?

This will become clearer in the next chapter once we go back to React Components! But mostly, closures are useful because we'd like to define a new function while keeping access to the variables defined in the outer function.
It may sound like closures is a "feature" that you decide to use, but in fact, a closure is what you get when you define a function.

Use unique variable names

The answer is that JavaScript will start looking in the current context before going outside the function. It keeps going until it reaches the global context, the `window` object.

In general, we recommend using unique variable names, so you don't have to worry about this. However, here's an example that uses the same variable name:

```
let test = 1;

function doSomething() {
    return test;
}

doSomething(); // 1
```

Now, let's define a variable `test` with a value of 2 inside the function:

```
let test = 1;

function doSomething() {
    let test = 2;
    return test;
}

doSomething(); // 2
```

Closures have more advanced use cases where the inner function captures the scope of its enclosing function and can access variables from that scope even after the outer function has completed execution.

A good example of this would be currying, which is a common pattern in functional programming. While this is outside of the scope of this course, here's a short example

first time, and then 10).

```
function curriedSum(a) {
    return function(b) {
        return a + b;
    }
}

// First usage
const sumFive = curriedSum(5);
sumFive(2); // 5 + 2 = 7
sumFive(10); // 5 + 10 = 15

// Second usage
const sumTen = curriedSum(10);
sumTen(3); // 10 + 3 = 13
sumTen(10); // 10 + 10 = 20
```

Recap

- A closure is where an inner function has access to the outer function's variables.
- Functions in JavaScript have access to their outer scope. This is called closures.
- Closures is what you get when you define a function; it's not something you have to "enable" or decide to use.
- Functions defined inside another function can use the variables defined in the outer function.

Was this helpful?

Chapter recap

Last updated June 2022

We learned about closures because as we work with **Events** & **State**, we will need to have event handlers that can access our state.

In a nutshell, we need to be able to write code like this:

```
import {useState} from "react";

function Counter() {
    const [count, setCount] = useState(0);

    function handleIncrementClick() {
        setCount(count + 1);
    }

    return (<>
        <div>{count} times clicked</div>
        <button onClick={handleIncrementClick}>Add 1</button>
    </>);
}
```

This will be fully explained in the next chapter, but for now, notice how **handleIncrementClick** is a function defined inside the **Counter** function.

And notice how the **handleIncrementClick** function has access to the `count` state variable because it's defined inside **Counter**.

This is why we need to know about closures :)

To recap **Closures** in one sentence:

It's when a function contains another function, the inner function gains access to the outer function's variables.

Chapter recap

- A closure is where an inner function has access to the outer function's variables.
- Functions defined inside another function can use the variables defined in the outer function.
- Closures are useful when working with React Components

Was this helpful?

Events

Last updated June 2022

We saw the basics of **click events** 2 chapters ago which allowed us to introduce the concept of hooks & state.

Now that we know about closures, it's time to dive deeper into **events** in React.

Attaching an event to a DOM element

In pure JavaScript, we use `element.addEventListener("click", () => {})`, however in React we use the following syntax in JSX:

`onClick={() => }`, for example: `<button onClick={() => console.log("I was clicked")}>Login</button>`.

It's important that it has to be `onClick` rather than `onclick` (the event name must be in lowerCamelCase).

Another common pitfall is when you forget the function definition, so `<button onClick={console.log("I was clicked")}>Login</button>` will log: `I was clicked` the moment the page has loaded and will not work on click.
That's because `console.log("I was clicked")` will execute on page load and will not attach to the `Click` event handler.

Accessibility

Technically speaking, you can add `onClick` to any DOM element, for example:

- p
- ul
- div
- button
- a

However, you should **only** add it to the `<button>` element for accessibility reasons. Accessibility is the practice of making your website accessible (usable) to all kinds of users regardless of abilities.
For example, some of your users may be using a **Screen Reader** to access your website. If you add an `onClick` to elements other than the `button`, they may not be able to click on those elements.

We'll cover that in more detail in a later chapter; for now, keep in mind that you should only use `onClick` on `<button>` elements.

Other types of events

We will be mostly working with `Click` events until we introduce the concept of **Forms** in a later chapter, but you can already see how to implement other types of events:

- "click" ⇒ `onClick`
- "keyup" ⇒ `onKeyUp`
- "keydown" ⇒ `onKeyDown`
- "change" ⇒ `onChange`

Notice that the event name is in lowerCamelCase preceded by `on`, meaning it's `onKeyPress` rather than `onkeypress`. The `k` has to be an uppercase `K` (the same for `P`).

```
<input type="text" placeholder="Enter your name" onChange={()
⇒ console.log('change!')} />
```

Recap

- `onClick` should only be used on `<button>` elements for better accessibility.
- `onEventName` is the general syntax for adding events to an element.

Was this helpful?

Named event handlers

Last updated June 2022

We've previously written our event handlers in an **inline** form which is when you directly assign a function (without a name) to the event, for example:

```
<button onClick={() => console.log("Logging
in..")}>Login</button>
```

The `() => console.log("Logging in..")` is an arrow function that is **inline** function which is convenient when you want to do a console.log (or a state change). But as we dive into more advanced concepts, we will have to have more advanced event handlers, so we'd like to define functions and give them names:

```
function handleLoginClick() {
    console.log("Logging in..");
};

<button onClick={handleLoginClick}>Login</button>
```

Notice how we defined `function handleLoginClick` and then **passed a reference of the function** to the `onClick` handler.

Common pitfalls

It's important to pass the *name of the function* rather than *calling the function*, so this will **not** work:

```
// this will NOT work
<button onClick={handleLoginClick()}>Login</button>
```

immediately.

Naming convention

We'd like to keep things organized as our components get more complicated.
So it's a good idea to have a naming convention for those event handlers.

Remember that React does not enforce these naming conventions, so your code will still work even if you don't follow them.

Here's the naming convention that we'll follow:

handleSubjectEvent

1. So the function's name always starts with `handle`.
2. Then it's followed by the subject: for example, `Login` for the Login button.
3. Then followed by the event type: `Click` for click.

So we end up with `handleLoginClick`.

Let's take other examples:

- `handleFormSubmit` for the submit event on a form
- `handleNameChange` for the change event on the name input
- `handleLogoutClick` for the click event on the Logout button

Why not handleEvent?

You might see `handleClick` or `handleSubmit`, which also works fine, except when you start having two events of the same type in the same component. Hence, you end up violating that naming convention.
This is why we prefer going with `handleSubjectEvent`.

Recap

- `<button onClick={handleLoginClick}>Login</button>` calls the `handleLoginClick` when the button is clicked.
- Use the `handleSubjectEvent` naming convention for event handlers

Was this helpful?

Events, State & Closures

Last updated January 2023

Now that we know about named events and closures, we can continue working with *state* inside our Components.

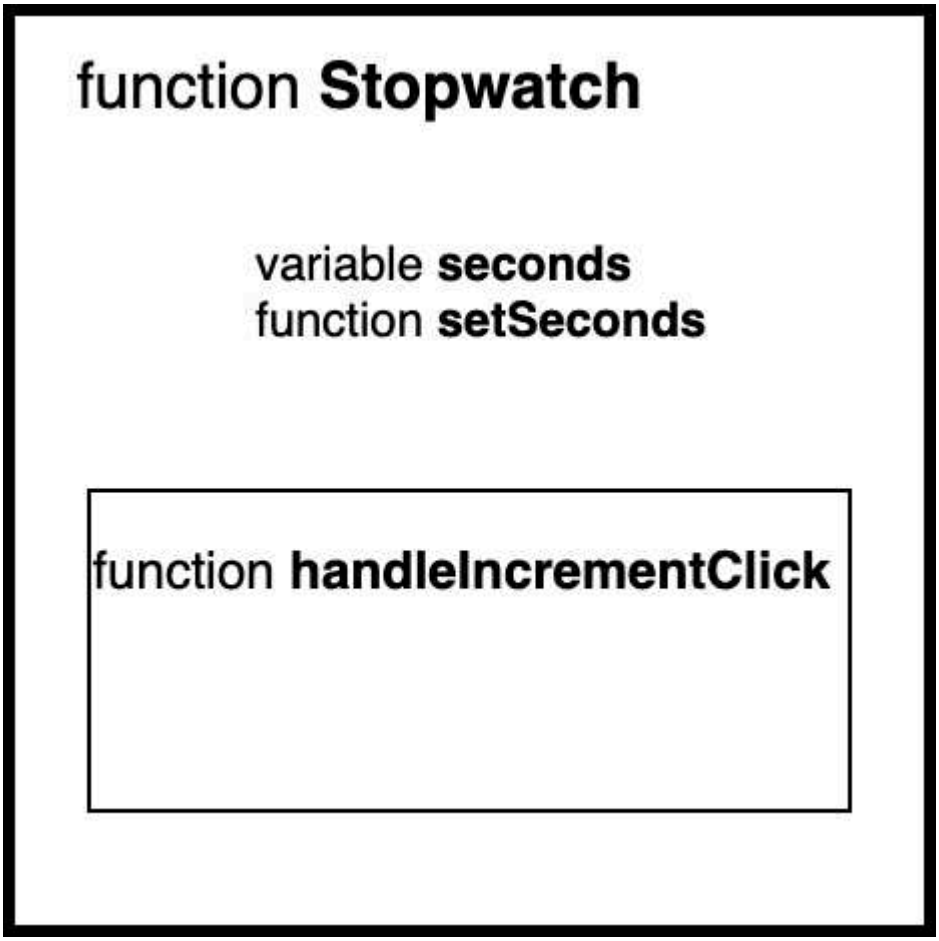
Let's move from inline event handler to a named one:

```
import {useState} from "react";

function Stopwatch() {
    const [seconds, setSeconds] = useState(0);

    return (<>
        <h2>{seconds}</h2>
        <button onClick={() => setSeconds(seconds + 1)}>Click to
add 1</button>
    </>);
}
```

Becomes:

```
import {useState} from "react";

function Stopwatch() {
    const [seconds, setSeconds] = useState(0);

    function handleIncrementClick() {
        setSeconds(seconds + 1);
    }

    return (<>
        <h2>{seconds}</h2>
        <button onClick={handleIncrementClick}>Click to add
1</button>
```

Notice how `handleIncrementClick` is defined *inside* the `Stopwatch` function. Because we have a *closure* here, it is possible for the `handleIncrementClick` to access the `seconds` variable because it's inside the scope of `Stopwatch`.



This is why the handleIncrementClick function should be defined **inside** the Stopwatch Component, or else it will not have access to its scope (so to the variables/functions such as `seconds` and `setSeconds`).

This will allow us to write more powerful Components in the next chapter! 😄

Recap

- When event handlers are defined inside the Component, they can use the state variables because of the closures concept.

Chapter recap

Last updated June 2022

Great job!

We'll learn in the next chapter about conditional state changes. See you there!

Chapter recap

- `onClick` should only be used on `<button>` elements for better accessibility.
- `onEventName` is the general syntax for adding events to an element.
- Moving from inline to named event handlers allows for more complexity inside the event handler.
- Use the `handleSubjectEvent` naming convention for event handlers
- When event handlers are defined inside the Component, they can use the state variables because of the closures concept.

Was this helpful?

Conditional state change

Last updated January 2023

The concept of conditional state change is not a *React concept* but more like a natural requirement that comes with several projects.

For example, when we built our Countdown component, it went from 10 to 9 when we clicked the `Countdown` button.
And then, from 9 to 8. We eventually were able to go below 0, which does not make sense from an end-user perspective; it should stop at 0.

This is where the concept of conditional state change comes in handy.

It's when you only update the state based on a certain condition.

Let's take an example where the `Add second` button keeps adding seconds until we hit 59.

```
import {useState} from "react";

function Clock() {
    const [seconds, setSeconds] = useState(0);

    function handleIncrementClick() {
        if (seconds < 59) {
            setSeconds(seconds + 1);
        }
    }

    return (<>
        <h1>{seconds} seconds</h1>
        <button onClick={handleIncrementClick}>Add</button>
    </>);
}
```

See how we're doing conditional state change based on the current state?

```
    setSeconds(seconds + 1);
  }
```

We only increment the seconds if `seconds < 59` returns true.
This means the last time we increment it is when seconds is 58, then seconds becomes 59, and then the `if` condition will not run anymore.

If this feels easy, it's because you've got a solid foundation regarding JSX, Components, Hooks & Events. Congratulations!

Important notes

Never wrap useState with an if condition.

Notice how we only wrapped `setSeconds` with an `if` condition, but the `useState` hook should never be wrapped with an if condition.

That's for two reasons:

1. The rules of hooks (will be explained in a few chapters). For now, keep in mind that React requires us to never wrap hooks with an if condition.
2. The `state` itself is not conditional. We always want to get the `seconds` state; it's the fact that the `setSeconds` call is conditional. Nevertheless we still destructure it from `useState` even if we don't end up calling it.

Recap

- Conditional state change is when you wrap the state **set** function with an if condition to satisfy some business logic.
- *NEVER* wrap `useState` with an if condition.

Was this helpful?

Conditional state change: Props

Last updated June 2022

What happens if we throw in Props into the mix?

For example, let's say we've got a Component that can increment a counter. Still, this component can also be enabled or disabled. When it's disabled, the counter won't increment.

```
import {useState} from "react";

function Counter(props) {
    const [counter, setCounter] = useState(0);

    function handleIncrementClick() {
        if (props.enabled){
            setCounter(counter + 1);
        }
    }

    return (<>
        <h2>{counter} times clicked</h2>
        <button onClick={handleIncrementClick}>Add 1</button>
    </>);
}

// Sample usages:
const element1 = <Counter enabled={true} />;
const element2 = <Counter enabled={false} />;
```

See how we can check the value of a **Prop** and then based on that change the state:

```
if (props.enabled) {
    setCounter(counter + 1);
}
```

Recap

- Props can be used to change the state conditionally.

Was this helpful?

Chapter recap

Last updated June 2022

Over the last few chapters, we started with the most basic example of the `Counter` Component and then made it more advanced as we learned more concepts.

A `Counter` is not precisely what you will be building in an actual project, so that's why we will have a Project where you can apply everything you learned so far.
But first, there's one more topic we'd like to cover before the Project, which is `Conditional Rendering`.

Chapter recap

- Conditional state change is when you wrap the state **set** function with an if condition to satisfy some business logic.
- *Never* wrap `useState` with an if condition.
- Props can be used to change the state conditionally.

Was this helpful?

Conditional rendering

Last updated June 2022

Conditional rendering is a topic that we've already covered however we'd like to take it a step further:

```
function WelcomeUser(props){
    if (props.user) {
        return <h1>Welcome {props.user}</h1>;
    } else {
        return <h1>Please login!</h1>;
    }
}
```

We can apply the same concept to components, so we end up having *conditional component rendering*:

```
import DarkTheme from "./DarkTheme.js"
import LightTheme from "./LightTheme.js"

function App(props) {
    if (props.theme === "dark") {
        return <DarkTheme />;
    }
    return <LightTheme />;
}
```

Assuming we've got two different components (`DarkTheme` & `LightTheme`), our `<App />` Component can render one of the two based on the result of an if condition.
This is called conditional rendering (more specifically, conditional component rendering).

This is useful when you'd like to render different components based on conditions.

Recap

- A component can render different components based on a specific condition.

Was this helpful?

Conditional rendering II

Last updated June 2022

There are other ways of conditional rendering, and these are primarily JavaScript features that you can use in React.

You are in no way obliged to use these methods, so we will never check if you're explicitly using them. Having said that, you will encounter these techniques in other people's code, so it's essential to understand how they work.

Storing elements in variables

This has already been covered in the JSX chapter, but we'd like to reiterate it. Since a JSX element such as `<h1 className="title">Hello</h1>` is just a call to `React.createElement( ... )` that returns an object, then you can store it in a variable.

This will allow you to re-use this variable if it gets repeated in a template or if you need to conditionally return it. For example:

```
function App(props) {
    const loginButton = <button>Login</button>;
    const signupButton = <button>Signup</button>;
    if (props.existingUser) {
        return <div className="app-container">{loginButton}
</div>;
    }
    return <div className="app-container">{signupButton}</div>;
}
```

Ternary operator

The ternary operator is a JavaScript syntax that lets you replace `if/else` with `? :`. Let's take an example:

```
    if (type === "admin") {
        return "Welcome admin";
    } else {
        return "Welcome user";
    }
}
```

Can be re-written as:

```
function getWelcomeMessage(type) {
    return (type === "admin") ? "Welcome admin" : "Welcome user";
}
```

The `? :` is the ternary operator where the expression after the `?` will be executed when the condition (`type === "admin"`) is `true` and the expression after the `:` will be executed when the condition is `false`.

This allows us to re-write the above App component:

```
function App(props) {
    const loginButton = <button>Login</button>;
    const signupButton = <button>Signup</button>;

    return <div className="app-container">{props.existingUser ?
loginButton : signupButton}</div>;
}
```

Note that, as experienced developers recommend, do **not** try and use the ternary operator everywhere.

You should always strive to write **readable** code, not necessarily fewer characters. In this example, the whole expression fits on a short line and is still readable, so `ternary` is acceptable. You can also keep the `if/else` condition as it's readable. So do not pressure yourself into using these techniques as readable code is your utmost priority as a developer.

Recap

- JSX elements can be stored in variables
- You can use the ternary operator as a shorter way to perform if conditions in JSX
- Always strive to write readable code rather than the shortest possible code.

Was this helpful?

Logical && operator

Last updated June 2022

The Logical && operator is also a JavaScript syntax that allows you to chain two expressions together.

The 2nd expression, will only execute if the first one is true, for example:

```
function getVotingMessage(age) {
    if (age ≥ 18) {
        return "You can vote";
    }
    return false;
}
```

Can be re-written as:

```
function getVotingMessage(age) {
    return (age ≥ 18) && "You can vote";
}
```

That's because when `age ≥ 18` returns false, the 2nd expression (after the &&) will NOT execute, so the function returns `false`.

On the other hand, when `age ≥ 18` returns true, the 2nd expression will execute and return `"You can vote"` which will then be returned from the function.

Logical && Operator in JSX

Here's an example of how you can use that in JSX:

```
function Counter(props) {
    if (props.count > 99) {
        return <>
```

```
        </>;
    }
    return <h1>You have {props.count} items</h1>;
}

// Sample usage
const element = <Counter count={15} />;
```

Can be re-written as:

```
function Counter(props) {
    return <>
        <h1>You have {props.count} items</h1>
        {props.count > 99 && <p>That is a lot of items.</p>}
    </>;
}
```

And now the `<p>That is a lot of items.</p>` will only be rendered when `props.count > 99` returns true!

Note that the below will **not** work as expected when `count` is 0:

```
function Counter(props) {
    return <>
        <h1>You have {props.count} items</h1>
        {props.count && <p>You have some items.</p>}
    </>;
}
```

When `count` is 0, you will get `0` to render on the screen. That's because `props.count` returns 0, which is a number, and React will render it because it's a number. For it to work as expected, you need to give an expression that returns a boolean (for example, `count > 0`). React will not render booleans.

Recap

- The logical && operator is a JavaScript feature that allows chaining two expressions. The 2nd expression will only execute if the first one returns true.
- Using the logical && operator in JSX makes it easier to render elements or components conditionally.

Was this helpful?

Chapter recap

Last updated July 2022

Nice!

Chapter recap

- A Component can render different Components based on a specific condition.
- JSX elements can be stored in variables
- You can use the ternary operator as a shorter way to perform if conditions in JSX
- Always strive to write readable code rather than the shortest possible code.
- The logical && operator is a JavaScript feature that allows chaining two expressions. The 2nd expression will only execute if the first one returns true.
- Using the logical && operator in JSX makes it easier to render elements or components conditionally.

Was this helpful?

Project III

Last updated May 2023

This is a very exciting project! You will build your first online store, which we will keep developing step by step until we have a proper Online Supermarket with Stripe integration.

State with Booleans

So far, we've used `useState` to keep track of state that represents numbers (such as a counter, a countdown, and seconds).
You can also use `state` to represent a boolean value (true/false).

Here's an example:

```
import {useState} from "react";

function App() {
    const [isFree, setIsFree] = useState(true);

    function handleUpgradeClick() {
        setIsFree(false);
```

```
    if (isFree) {
        return <button onClick=
{handleUpgradeClick}>Upgrade</button>;
    }
    return <h2>Welcome Paid user!</h2>;
}
```

In this example, `isFree` defaults to `true` because that's the default value that we passed to `useState`. Then when the user clicks on the `button`, we set it to `false`.

App component

The `index.js` will have an `App` component.
This is the only component being passed to ReactDOM's `render` function and will be responsible for showing the User Components based on some factors (for example, state).

This is how most React apps are built; if you use Vite (which we will see in chapter 30), you will always get the `App` component by default.

Goal of Project III

The goal of Project III is to solidify your knowledge with the concepts we just learned, ranging all the way from basic JSX to the five last concepts we learned:

- State
- Closures
- Events
- Conditional rendering
- Conditional state change

All while building a real-life project.

Enjoy!

Project recap

Last updated June 2022

Congratulations on completing Project III!

Let's take a step back and see how React is helping us build Applications:

Separation of concern

Rather than having our application logic scattered around or all in the same file, we can now look for our application logic in specific files:

- The **Product** logic is in the **Product.js**
- The **StoreFront** logic is in the **StoreFront.js**

This makes it easier to maintain in the future!

No need to worry about DOM updates

We did not have to use `document.querySelector` or `.addEventListener` to make our app dynamic.
We define the state & events, and React takes care of re-rendering the parts that have changed.

Code reuse

What if we need to render the **Product** somewhere else? Maybe on another page?
We can import and render the `<Product details={ ... } />`, and it will work!

Current limitations

You may have noticed that this project had some limitations, for example:

2. We duplicated the `<Product />` twice because we don't know how to map through an array in JSX yet.
3. The `<Product />` Component became relatively big. We will learn how to break it down into smaller pieces.

These are some of the concepts that we will be covering next!

Was this helpful?

Multiple states

Last updated September 2022

In this chapter, we'll look at *multiple states* in the same component as well as the rules of hooks (covered in the next lesson).

It is possible to create multiple states in the same component by having multiple calls to `useState`.
It's important, however, to keep those state calls **at the top of the component**, and you should **never** wrap them with an if condition.
We will explain why in the next lesson.

```
import {useState} from "react";

function Counter() {
    const [count, setCount] = useState(0);
    const [isLocked, setIsLocked] = useState(false);

    // ...
}
```

Nothing exceptional, as you can see, but this will be used to explain the rules of hooks in the next lesson!

Recap

- You can use multiple states in the same component by having multiple (and separate) `useState` calls.

Rules of Hooks

Last updated June 2022

Hooks may seem like magic, but we will demystify them along the way.

For hooks to work correctly, you have to follow two rules:

Rule #1: Only call Hooks from React functions

So far, the way we've taught this course, you can't really break this rule. So let's not worry about it for now.
All you have to know for now is that you should only call hooks in **Components**.

Rule #2: Only call Hooks at the Top Level

and never call hooks inside loops, conditions, or nested functions.

Since the beginning of the Hooks chapter, we've asked you to declare the hooks at the top of the function. In the conditional state change chapter, we've also learned to **never** wrap the `useState` call with an if condition.

The reason behind it is that React needs to have the same **order** of hooks with **every** function call. For example:

```
import {useState} from "react";

function App() {
    const [count, setCount] = useState(5);
    const [lives, setLives] = useState(3);

    return <button onClick={() => setCount(count - 1)}>Click
```

The first time the function is called, this is the order in which the hooks are called:

```
useState(5);
useState(3);
```

This will allow React to create two separate state variables, the first one defaulting to 5 and the second one defaulting to 3.

The next time the Component is called, for example, when the user clicks on the button, the order will be the same:

```
useState(5);
useState(3);
```

However, the first state variable will now be 4 because we decremented it because of the button click. In contrast, the second state variable remains 3 because we have not changed.

Having the same order will allow React to keep track of these 2 state variables separately and make sure to update the proper state value when it changes.
So, in summary: React relies on the order in which Hooks are called.

This is the reason why you're not allowed to wrap the hooks with an `if` condition, a loop, or a nested function.

```
import {useState} from "react";

function App(props) {
    // this will NOT work
    if (some_condition){
        const [count, useCount] = useState(0);
    }
}
```

component will result in the same order (and same amount) of state calls.

This is why React asks you to put all your hooks calls at the top. That makes it easier for you to remember that they should not be wrapped with loops, if conditions, and nested functions.

Recap

- For hooks to work correctly, you have to follow two rules.
- Rules of hooks #1: Only call Hooks from React functions
- Rules of hooks #2: Only call Hooks at the Top Level and never call hooks inside loops, conditions, or nested functions.
- React relies on the order in which Hooks are called

Was this helpful?

Chapter recap

Last updated June 2022

We've seen so far how to work with state that represents **numbers** and **booleans**.

We can also create state that represents strings:

```
import {useState} from "react";

function App() {
    const [name, setName] = useState("");

    return <input type="text" />
}
```

We will dive deeper into this once we start working with **forms**.

Next, we will learn how to work with state representing **arrays**, **objects**, and **arrays of objects**, but first, we have to cover **immutability** which is a requirement and a fundamental concept in React.

Chapter recap

- You can use multiple states in the same component by having multiple (and separate) `useState` calls.
- For hooks to work correctly, you have to follow two rules.
- Rules of hooks #1: Only call Hooks from React functions
- Rules of hooks #2: Only call Hooks at the Top Level and never call hooks inside loops, conditions, or nested functions.
- React relies on the order in which Hooks are called

Arrays & Objects

Last updated June 2022

This chapter is meant to prepare you for an important concept in React that is influenced by how JavaScript works.

Let's start by comparing some numbers, strings & booleans:

```
1 === 1; //true
27 === 27; //true
"hello world" === "hello world"; //true
"welcome" === "welcome"; //true
true === true; //true
false === false; //true (because they're the same)
```

There's nothing special here. We're comparing two values that are exactly the same, so the result is `true`.

Now let's try with arrays and objects:

```
[] === []; //false
{} === {}; //false
[10] === [10]; //false
{key: "something"} === {key: "something"}; //false
```

Note: you will still get `false` even if you used the `==` instead of triple equal `===`.
That's weird.. why?

Arrays & Objects

Arrays & Objects are both considered **objects** in JavaScript.
When you write `[]`, it's the same as creating a new instance of `Array`.
When you write `{}`, it's the same as creating a new instance of `Object`.

```
new Object(); //creates {}
```

Let's go back and change the arrays & objects in the previous examples to this new syntax:

```
new Array() === new Array(); //false
new Object() === new Object(); //false

const arr1 = new Array();
arr1.push(10);
const arr2 = new Array();
arr2.push(10);
arr1 === arr2; //false

const obj1 = new Object();
obj1.key = "something";
const obj2 = new Object();
obj2.key = "something";
obj1 === obj2; //false
```

This should make the reason why they're not equal a bit clearer.
`new Array()` creates a new instance of *array*.
Every time you call `new Array()`, you get a new instance, which means `new Array()` is certainly not the same as `new Array()` because they both are different instances.

So with Arrays & Objects, we need another approach for comparing equality from a values perspective.
As in, we expected it `[] === []` to be true because they are both empty arrays, but the way JavaScript works is different as it's checking if they are the same instance.

Object assignment

There's another *feature* in JavaScript that may seem counterintuitive that you must know about. That is what happens when we assign a variable to an existing object (or array).

```
const firstArray = [];
const secondArray = firstArray; // secondArray now points to
firstArray
console.log(firstArray); // []
console.log(secondArray); // []

firstArray.push(10);
console.log(firstArray); // [10]
console.log(secondArray); // [10]
```

You may be surprised: why does `secondArray` also contain `10` now? Well that's because when we created `secondArray = firstArray`, we are **not** copying firstArray, but instead creating a reference to it.

Which means `firstArray` and `secondArray` are now referring to the same value.

Technically we say they are referencing the same place in the memory.

So whenever you change the value, firstArray and secondArray will both read the same value that has been updated.

Recap

- Arrays are objects in JavaScript.
- `[] === []` is the same as `new Array() === new Array()`
- `{} === {}` is the same as `new Object() === new Object()`
- `[] === []` is `false` because it's comparing 2 different instances of arrays
- `{} === {}` is `false` because it's comparing 2 different instances of objects
- When you assign a variable to an object or array, it does NOT copy it. It will reference its value.

Was this helpful?

Mutations

Last updated November 2023

Going back to the previous example, we're going to add `console.log(firstArray === secondArray);` comparisons throughout our code:

```
const firstArray = [];
const secondArray = firstArray; // secondArray now points to
firstArray
console.log(firstArray === secondArray); //true and expected

firstArray.push(10);
console.log(firstArray === secondArray); //true but unexpected
```

We say that `.push` is a method that *mutates* the array, which means it's a method that changes the array.
Instead, in the next chapter, we're going to learn several methods we can use that do not mutate the array but rather create a new copy of it.
And when we create a new copy of it, then the second comparison would turn out to be `false` (which is something we'd expect).

Why are we doing all of this? We will explain that in the next lesson. In short, React needs a way to know in an efficient manner if you've changed the state (when it's an array or object).

What is Immutability?

An immutable object is one that cannot be changed. As we cannot (or sometimes, should not) modify an immutable object, we create an entirely new object based on the original, leaving the original object unchanged.

When working with arrays and objects in React, we will walk you through this fundamental concept in the following few chapters.

Deep equal is when you compare two objects based on their values.
With a deep equal, `[]` would be equal to `[]`. Same for `{key: "something"}` and `{key: "something"}`.
However, JavaScript does NOT have a built-in method for deep equal. Recently, browsers have shipped a `structuredClone` method that lets you deep clone an object. However, this would **not** be efficient to use inside React, as you will see in the next lesson that React will need to make comparisons **very frequently** as part of its Virtual DOM to know which components need to be re-rendered.

Recap

- `Array.prototype.push` mutates the array.
- An immutable object is one that cannot be changed. As we cannot (or sometimes, should not) modify an immutable object, we create an entirely new object based on the original, leaving the original object unchanged.

Was this helpful?

Why React requires immutability

Last updated November 2023

Let's see why React requires immutability when it comes to Arrays & Objects. Let's say you wrote this Component:

```
import {useState} from "react";

function App() {
  const [data, setData] = useState([]);

  function handleAddClick() {
      data.push(10)
      setData(data);
  }

  return <button onClick={handleAddClick}>Add 10</button>;
}
```

This is not the correct way to add 10 to the **data** state (which is an array). But let's see why and what happens behind the scene.

When you call with an empty array, `const [data, setData] = useState([])` it's going to create a state variable with the value `[]`.

Then the setData function will take the `newState` (which is the new value of the state) and then check if it has changed. It will ask ReactDOM to re-render this Component if it has changed.

If we were to write a simplified `setData` function, it would look like this:

```
let state = []; //created by `useState([])`

// newState is the result of `data.push(10)` on <button /> click
function setData(newState) {
```

```
      return false;
    }
    // store the newState for the next time the user calls
  setData()
    state = newState;
    // Ask ReactDOM to re-render
  }
```

Notice how React compares `state === newState`? This allows React to know if the state has changed.
If `state === newState` is `true`, it means the state has NOT changed, which means there's no need to re-render the Component.
But when `state === newState` is `false`, it means that the state has changed, and React **has** to re-render the Component using ReactDOM.

This is a detailed explanation of the Reconciliation topic we saw earlier in the course.

What happens without *immutability*?

In the previous example, we've used `data.push(10)` which *mutates* the array. If we write all the operations line by line, they would look as follows:

```
let state = []; //from useState([])
let newState = state;
state.push(10);

state === newState; //true, whereas it should have been false
```

So because we mutated the array with `.push`, React will think that we have NOT changed the state and thus will end up NOT re-rendering our Component.
And this is why React requires immutability.
So the only way for `state === newState` to be `false` when we modify the array is by returning a completely **new array**. That's going to be the topic for the next chapter 😄.

equal check would be way too slow (as the number of components in your app increases).

This is why every time you have a state of arrays or objects, they must be immutable.

Recap

- React stores the current state and compares it to the new state with the triple equal comparison.
- If you mutate the new state, React has no way to know it has changed.
- React re-renders your Component automatically when `state === newState` returns `false`.
- Every time you have a state of arrays or objects, they must be immutable.

Was this helpful?

Chapter recap

Last updated November 2023

Immutability is such an important concept that we decided to cover it in several chapters.
The chapter you just completed introduced the concept of immutability while explaining why it's needed.
Next, we'll learn how to make changes to arrays while keeping them immutable (so we will create new copies of them).
Then we'll be able to go back to React-related challenges where we use state with arrays.
Then we're going to do the same thing for Objects. Exciting!

Is immutability really worth it?

All the work around immutability **is** what's letting React handle thousands of Components on your App, so yes, it's definitely worth it!

Chapter recap

- Arrays are objects in JavaScript.
- `[] === []` is `false` because it's comparing 2 different instances of arrays
- `{} === {}` is `false` because it's comparing 2 different instances of objects
- When you assign a variable to an object or array, it does NOT copy it. It will reference its value.
- `Array.prototype.push` mutates the array.
- An immutable object is one that cannot be changed. As we cannot (or sometimes, should not) modify an immutable object, we create an entirely new object based on the original, leaving the original object unchanged.
- React stores the current state and compares it to the new state with the triple equal comparison.

`newState` returns `false`.

- Every time you have a state of arrays or objects, they must be immutable.

Was this helpful?

Array Immutability

Last updated June 2022

Now that we know that Arrays & Objects require special attention regarding immutability, we will have dedicated chapters to explain how to perform immutable operations. This chapter will cover arrays!

We'll cover how to add, update & remove items to/from an array in an immutable way.

Spread syntax recap

Before starting, we need to recap the spread syntax that is part of ES2015 (or ES6).

```
const numbers = [1, 2, 3];
const grades = [ ... numbers];
console.log(grades); // [1, 2, 3] (shallow copy)
```

`... numbers` will spread the numbers. This means it will remove the array from them, so they become `1, 2, 3`.
So then `[ ... numbers]` is creating a new copy of the array with the same values because it ends up being equivalent to `[1, 2, 3]`.
We call this a *shallow copy* because it's a copy of the items inside the array (but down to 1 level - don't worry about this detail for now).

Concatenate arrays

You can use the spread syntax to concatenate arrays, for example:

```
const winners = ["Jane", "Bob"];
const losers = ["Ronald", "Kevin"];

const players = [ ... winners,  ... losers];
console.log(players); // ['Jane', 'Bob', 'Ronald', 'Kevin']
```

Add an item to an array (immutable)

So how do we add an item to an array in an immutable way?

We can't use `.push()` because push will mutate the array.
Instead, we have to make a shallow copy and then insert the new item in a new array.
Here's what it looks like:

```
const numbers = [1, 2, 3];
const result = [...numbers, 4];
console.log(result); //[1, 2 ,3 ,4]
```

We spread the values of the `numbers` array and add 4 afterward. The new array will contain `1, 2, 3, 4`.
This is immutable because we are NOT changing the `numbers` array but rather creating a new copy with the extra number in it.

Recap

- The spread operator in an array creates a shallow copy.
- `[...existingArray, "newValue"]` adds `newValue` to the array in an immutable way because it creates a copy of it.

Was this helpful?

Update & Remove

Last updated June 2022

In this lesson, we're going to learn how to update & remove items from an array in an immutable way.

Update item(s) (immutable)

To update one or more items from an array, you can use the *.map* method to return a copy of the array while modifying one or more items. For example:

```
const grades = [10, 20, 18, 14];
// change 18 to 17
const updatedGrades = grades.map(grade => {
    if (grade === 18){
        return 17;
    }
    // in all other cases, keep it as it was
    return grade;
});
console.log(updatedGrades); //[10, 20, 17, 14]
```

You can also update multiple elements, for example add 1 to all of the grades that are under 10:

```
const grades = [10, 8, 9, 4, 16];
// add 1 to all grades below 10
const updatedGrades = grades.map(grade => {
    if (grade < 10){
        return grade + 1;
    }
    // in all other cases, keep it as it was
    return grade;
```

Remove item(s) (immutable)

You can remove items in an immutable way by using the *slice* method (pay attention that it's *slice*, not *splice*). *slice* is immutable, whereas *splice* will mutate the array.

```
const grades = [10, 8, 9, 4, 16];

// remove the first grade
// think of it as: get all grades except the first one
const subset1 = grades.slice(1); //start from position 1
console.log(subset1); // [8, 9, 4, 16]

// remove the last 2 grades
// think of it as: get all grades except the last 2
// so start from 0 and stop after 5 - 2 = 3 items
const subset2 = grades.slice(0, grades.length - 2);
console.log(subset2); // [10, 8, 9]
```

Sometimes it's easier to use *.filter*, which will return a subset of the original array based on a condition, for example:

```
const grades = [10, 8, 9, 4, 16];

// return all grades ≥ 10
const subset1 = grades.filter(grade => grade ≥ 10);
console.log(subset1); // [10, 16]

// remove the 2nd grade
const subset2 = grades.filter(grade => grade ≠ 8);
console.log(subset2); // [10, 9, 4, 16]
```

Note that the 2nd example will skip *all* grades that are 8. If you only want to skip the 2nd item, then you can use the 2nd argument that you receive with `filter`, which is

```
const grades = [10, 8, 9, 4, 16];

const subset = grades.filter((grade, index) => index !== 1);
console.log(subset); // [10, 9, 4, 16];
```

Recap

- `.map` helps you update an array in an immutable way.
- `.slice` is immutable, whereas `.splice` mutates the array.
- `.slice` helps you remove items from an array in an immutable way.
- `.filter` helps you remove items from an array in an immutable way.

Was this helpful?

Chapter recap

Last updated July 2023

Nice work!

Now that we know how to manipulate arrays in an immutable way, we're going to go back into React land and learn how to use state with arrays to create meaningful components!

Chapter recap

- The spread operator in an array creates a shallow copy.
- `[...anArray, "newValue"]` adds `newValue` to the array in an immutable way because it creates a copy of it.
- **.map** helps you update an array in an immutable way.
- **.slice** is immutable, whereas **.splice** mutates the array.
- **.slice** helps you remove items from an array in an immutable way.
- **.filter** helps you remove items from an array in an immutable way.

Was this helpful?

Arrays in JSX

Last updated June 2022

The goal of this chapter is to work with *state* that is an array. Remember that arrays should be immutable in React.

When working with Arrays, you will often need to iterate over them. In Project III, we did not know how to iterate over our two products, so instead, we opted to render them by the index. There's definitely a better way!

Let's say we've got an array of `grades` and we want to create a `ul` and one `li` for every grade:

```
function Grades(){
    const grades = [8, 18, 10, 7, 14];

    // this will generate a warning (keep reading)
    return <ul>
        { grades.map(grade => <li>{grade}</li>) }
    </ul>;
}
```

This will render:

```
<ul>
    <li>8</li>
    <li>18</li>
    <li>10</li>
    <li>7</li>
    <li>14</li>
</ul>
```

Keep in mind that this works because we're using the implicit return inside the map callback. Alternatively, if you decide to write it on multiple lines, then the `return` becomes mandatory:

```
    const grades = [8, 18, 10, 7, 14];

    // this will generate a warning (keep reading)
    return <ul>
        {
            grades.map(grade => {
                return <li>{grade}</li>
            })
        }
    </ul>;
}
```

You can also decide to store the result of a map in a variable, and then refer to it in JSX:

```
function Grades(){
    const grades = [8, 18, 10, 7, 14];

    const listItems = grades.map(grade => <li>{grade}</li>);

    // this will generate a warning (keep reading)
    return <ul>
        {listItems}
    </ul>;
}
```

Why will all of these examples generate a warning? We'll find out in the next lesson.

Recap

- `.map` can be used inside JSX to loop through arrays

Was this helpful?

Providing a key

Last updated April 2023

Every time you have a `map` in JSX, you need to provide a `key` or else you will get a warning (for performance reasons):

```
function Grades() {
    const grades = [8, 18, 10, 10];

    return <ul>{
        grades.map((grade, index) => <li key={index}>{grade}
</li>)
    }</ul>;
}
```

The `key` should be a unique representation of the single item inside the map. However, because the grades are not necessarily unique, we can use the `index` provided by the `.map` method.

The index starts from 0 and is incremented by 1 for every iteration.

If the array had unique items, then the key would be that item:

```
function Users() {
    // collection of user ids
    const users = [1, 10, 3, 4, 13];

    return <ul>{
        users.map(user => <li key={user}>{user}</li>)
    }</ul>;
}
```

So using the `index` as a key should be the last resort. It's okay for now since we're only working with arrays of strings and arrays of numbers; for now, then it's fine. When we learn about arrays of objects, then we'll let you know what the `key` should be.

Now let's say one of array items changes, React then needs to know which `li` to update; thus, it requires a unique `key` so that it can only update that item without having to remove all the `lis` and render them again.

The same applies to items that need to be added or removed.

For example:

```
import {useState} from "react";

function Users() {
    // collection of user ids
    const [users, setUsers] = useState([4, 8, 10, 17, 13]);

    function handleRemoveClick() {
        // remember array immutability ;)
        setUsers(users.slice(1));
    }

    return <>
        <ul>{
            users.map(user => <li key={user}>{user}</li>)
        }</ul>
        <button onClick={handleRemoveClick}>Remove first
user</button>
    </>;
}
```

Having the `key={user}` where `user` is unique will allow React (specifically the Virtual DOM) to understand that since the new rendered elements do not contain an `<li key="4">`, then this `li` needs to be removed. All the rest will stay intact.

Without this `key`, React would have to remove all the `lis` and then add them again, which would slow down your app.

Recap

- Whenever you iterate with `.map` in JSX, you should provide a `key`.
- The `key` should be unique. As a last resort, you can use the `index` of the map iteration (which is the 2nd argument)
- The `key` allows React to update the DOM efficiently with the least amount of operations.

Was this helpful?

Chapter recap

Last updated June 2022

Awesome!

Chapter recap

- `.map` can be used inside JSX to loop through arrays
- Whenever you iterate with `.map` in JSX, you should provide a `key`.
- The `key` should be unique. As a last resort, you can use the `index` of the map iteration (which is the 2nd argument)
- The `key` allows React to update the DOM efficiently with the least amount of operations.

Was this helpful?

Add & Replace

Last updated June 2022

Now that we know how to modify arrays immutably, object immutability becomes easier as it also relies on the spread (`...`) operator.

Add key/value immutably

Let's start with the example that **mutates** the object:

```
const data = {
    id: 1,
    name: "Sam"
}

//BAD: mutates
data.age = 18;
console.log(data); // {id: 1, name: "Sam", age: 18}
```

Instead, we need to make a copy of that object using the spread operator: `{ ... data}`. This creates a **new** object with the same keys and values, and then we can add the key/value pair that we want to add next to it:

```
const data = {
    id: 1,
    name: "Sam"
}

//GOOD: immutable
const newObj = { ... data, age: 18}
console.log(newObj); // {id: 1, name: "Sam", age: 18}
```

Here's the approach that **mutates** the object:

```
const data = {
    id: 1,
    age: 19
}

// BAD: mutates
data.age = 20;
console.log(data); // {id: 1, age: 20}
```

Instead, we need to create a new copy of that object with `{ ... data}` and then merge it with the new same key but with a different value:

```
const data = {
    id: 1,
    age: 19
}

// GOOD: immutable
const newObj = { ... data, age: 20};
console.log(newObj); // {id: 1, age: 20}
console.log(data); // original object did not change {id: 1, age:
19}
```

This works because the `age: 20` part from `{ ... data, age: 20}` will override the existing `age`.

Order matters

It's important to note that when you want to replace, the new values should be **after** the copy of the old object.

This will allow overriding the old values into the new ones.

This will **NOT** work:

```
    id: 1,
    age: 19
}
const wrongNewObj = {age: 20, ...data};
```

because the `age: 19` from the `data` object ends up overriding the `age: 20`.

Simply switching it to `{ ... data, age: 20}` will make it work.

Recap

- `{...obj}` allows you to create a new copy of the existing object, which is useful for immutable operations.
- `{...obj, newKey: 'value'}` allows you to immutably add `newKey: 'value'` to an object `obj`.
- `{...obj, existingKey: 'newValue'}` allows you to immutably change the `existingKey` to `newValue`.

Was this helpful?

Remove key/value pair

Last updated August 2022

In order to immutably remove a key/value pair from an object, we will also rely on the spread operator but in a slightly different approach.

First let's start with the approach that **mutates** the object:

```
const obj = {
    id: 1,
    title: "Harry potter",
    year: 2017,
    rating: 4.5
}

// BAD: mutates
delete obj.year;
console.log(obj); // { id: 1, title: "Harry potter", rating: 4.5}
```

In order to delete the `year` in an **immutable** way, we will have to use 2 JavaScript features: *object destructuring* and the *spread operator*:

```
const obj = {
    id: 1,
    title: "Harry potter",
    year: 2017,
    rating: 4.5
}

// GOOD: immutable
const {year, ...rest} = obj;
console.log(rest); // { id: 1, title: "Harry potter", rating:
4.5}
```

the key year from obj.

This is similar to reading `obj.year`.

However, notice how we ask JavaScript to destructure the rest of the object with `... rest`; this means combining *all* the other key/values in a new object called `rest`.

So we end up with `rest`, an immutable copy of obj excluding the `year`!

Let's take another example:

```
const data = {
    id: 1,
    grades: [10, 14, 8],
    count: 3,
    course: "react"
}

// Immutably remove: grades & course
const {grades, course, ...newData} = data;
console.log(newData); //{id: 1, count: 3}
```

We started by destructuring the `grades` and the `course` and then put all the rest in a new object called `newData`!

Recap

- `const {keyToRemove, ...rest} = obj` is an easy way to immutably remove `keyToRemove` from an object.

Was this helpful?

Chapter recap

Last updated August 2022

Nice work!

Chapter recap

- `{...obj}` allows you to create a new copy of the existing object which is useful for immutable operations.
- `{...obj, newKey: 'value'}` allows you to immutably add `newKey: 'value'` to an object `obj`.
- `{...obj, existingKey: 'newValue'}` allows you to immutably change the `existingKey` to `newValue`.
- `const {keyToRemove, ...rest} = obj` is an easy way to immutably remove `keyToRemove` from an object.

Was this helpful?

State with Objects

Last updated February 2023

This chapter aims to work with *state* that is an object.

Don't forget that whenever you want to modify an object, it has to be in an immutable way!

Default value

Like any other type of state, you need to give it a default value.

The most common form you'll see is defaulting to an empty object:

```
import {useState} from "react";

function App() {
    const [data, setData] = useState({});
}
```

You can also default to any other object, for example:

```
{
    darkMode: false
}
```

Tip: inverting a boolean

Here's a quick tip that might come in handy. If you'd like to invert a boolean (as in go from `false` to `true` and from `true` to `false`), then you can use the `Logical Not` operator: `!` (also known as the Bang operator).

Here's how:

```
const booleanValue2 = false;
const inverted2 = !booleanValue2; // true
```

Loop through objects in JSX

Sometimes it may be useful to loop through an object. Note that this is not as common as looping through an array, but in case you need to, here's how you can do it:

```
function App() {
    const settings = {
        title: "Blog",
        theme: "dark"
    }

    return <ul>{
        Object.entries(settings).map(item => {
            return <li key={item[0]}>{item[0]} with value
{item[1]}</li>
        })
    }</ul>;
}
```

The resulting JSX will be:

```
<ul>
    <li key="title">title with value Blog</li>
    <li key="theme">theme with value dark</li>
</ul>
```

This works because `Object.entries(settings)` returns an array that looks like this:

```
[
    ["title", "Blog"],
```

Recap

- Remember object immutability when working with state that is an object.
- You can invert a boolean by preceding it with the logical Not operator.
- `!true` gives `false` and `!false` gives `true`.

Was this helpful?

Chapter recap

Last updated June 2022

Great job going through the immutability chapters!

We're still going to cover arrays of objects in a future project, but for now, we're going back to diving into React as the next chapter is about *Forms*!

Recap

- Remember object immutability when working with state that is an object.
- You can invert a boolean by preceding it with the logical Not operator.
- `!true` gives `false` and `!false` gives `true`.

Was this helpful?

Default value

Last updated June 2022

This chapter will cover how to work with forms in React.
We'll start with the basics and build on them step by step.

value attribute in HTML

In HTML, we're used to giving a default value for a textbox by specifying the `value` attribute, for example:

```
<!— HTML example —>
<input type="text" name="address" value="Amsterdam">
```

This renders a textbox with the value `Amsterdam` and the user will be able to edit it.

However, the behavior in React is different.

defaultValue to the rescue

When you set the `value` on an `input` in React, its value will never change (unless you specify an `onChange` handler which we'll learn about in the next lesson).

Thus the below JSX is something you'll almost never want:

```
<input type="text" name="address" value="Amsterdam" />
```

Notice the self-closing tag syntax because we're in JSX; this is mandatory.

The end result of this element is a textbox with the value `Amsterdam` that cannot be changed, so effectively, it's a read-only textbox.

```
<input type="text" name="address" defaultValue="Amsterdam" />
```

In the next lessons, we'll learn how to read the value that's written by the user and how to keep track of it in a `state` variable.

Recap

- Setting a `value` on an input without an `onChange` handler will create a *read-only* input.
- If you want to provide a default value, then use the `defaultValue` attribute.

Was this helpful?

Get input value

Last updated February 2023

To get the text written by the user, we use an `onChange` event handler on the `<input />`:

```
function handleAddressChange(event) {
    // ...
}

<input type="text" name="address" onChange={handleAddressChange}
/>
```

The `handleAddressChange` function will be called every time the user enters a new character, removes one, or makes any edit to the textbox. It will fire every time the input `changes`.

The event argument

The function passed to `onChange` will receive an `event` argument, similarly to how you'd expect it when working with the DOM.
However, this `event` is technically a *synthetic event* which we'll cover later in the course.

You can read the value written by the user by accessing: `event.target.value`:

```
function handleAddressChange(event) {
    console.log(event.target.value);
}

<input type="text" name="address" onChange={handleAddressChange}
/>
```

Because it's an input, you read what's written inside it by accessing the `value` property.

target vs currentTarget

If you've been writing plain JavaScript code before, you may be familiar with a recommendation to use `currentTarget` over `target` (because `currentTarget` will always refer to the element on which you called `addEventListener` whereas `target` will depend on where exactly the user clicks).

When it comes to React's onChange, there is no difference between `target` and `currentTarget` because there's only one element with no children. So, both values will resolve to the same element.

You will see developers using `target`, which we also recommend.

Inline functions

It's common to use inline functions with forms as you often need to have a small event handler, here's how it would look like:

```
<input type="text" name="address" onChange={event =>
console.log(event.target.value)} />
```

Recap

- onChange on an input will fire every time the user makes a change to that textbox.
- You can read the user's text with `event.target.value`.

Was this helpful?

Controlled components

Last updated June 2022

The previous two lessons prepared you to understand the concept of *controlled components* in React.

What is a controlled component?

A controlled component is when you keep track of an `input`'s value as `state` and update it whenever it changes.

This is useful almost every time you have an `<input />` (or even a `<select />` or `<textarea />` elements) as it lets you get the value written by the user as well as automatically change its value when the state changes.

Here's how you create a controlled component:

1. We start by creating a `state` variable to store the value
2. This state will have a default value of an empty string `""`
3. We set the value of the input to that `state` variable
4. We update the state every time it changes

```
import {useState} from "react";

function App() {
    const [address, setAddress] = useState("");

    return <input type="text" value={address} onChange={event ⇒
setAddress(event.target.value)} />;
}
```

Let's re-iterate over the steps:

1. We start by creating the `address` state

3. We set the `<input />`'s `value={address}`
4. We update the state every time it changes: `onChange={event => setAddress(event.target.value)}`

Now every time the user changes the content of the textbox, you will be able to get the most up-to-date version by accessing the `address` state.
And if you want to change the value of `address` to another value, then with a `setAddress(" .. ")` call, you will change it and update the textbox's value.

This is a Controlled Component :)

Providing a default value

If you'd like a default value other than an empty string, that's also possible. You can do so by updating the default value of `useState`:

```
import {useState} from "react";

function App() {
    const [address, setAddress] = useState("Amsterdam");

    return <input type="text" value={address} onChange={event => setAddress(event.target.value)} />;
}
```

You don't need the `defaultValue` attribute anymore because this is a controlled component.

Recap

- A controlled component is when you keep track of an `input`'s value as `state` and update it whenever it changes.
- To make an input controlled, you have to assign its value to a state variable and update that state variable whenever the input changes.

```
{event => setState(event.target.value)}
```

Was this helpful?

Submitting forms

Last updated June 2022

Forms in real projects will most likely have more than one field, and it's common to encounter `<select />` or `<textarea />` elements.

Luckily the process of making those elements controlled is the same as for the `<input />`.

Select

```
import {useState} from "react";

function App() {
    const [country, setCountry] = useState("");

    return <select value={country} onChange={e =>
setCountry(e.target.value)}>
        <option>Country</option>
        <option value="netherlands">Netherlands</option>
        <option value="belgium">Belgium</option>
        <option value="france">France</option>
    </select>
}
```

Note: we've renamed `event` to `e` because it's shorter.

Textarea

```
import {useState} from "react";

function App() {
    const [comment, setComment] = useState("");

    return <textarea value={comment} onChange={e =>
```

Multiple inputs

When working with multiple inputs and assuming no validations (which we have not learned about yet), we recommend you stick to inline functions as the onChange's only goal will be to set the state.

We use a different state variable for every textbox:

```
import {useState} from "react";

function Form() {
    const [name, setName] = useState("");
    const [address, setAddress] = useState("");

    return <form>
        <input type="text" name="name" value={name} onChange={e
⇒ setName(e.target.value)} />
        <input type="text" name="address" value={address}
onChange={e ⇒ setAddress(e.target.value)} />
    </form>
}
```

Submitting a form

`Form` elements in React behave the same way as they would without React, meaning that when you `submit` a form, it will send the data to the current page, which ends up reloading it.

When working with fetch (which we'll cover soon), you'd need to prevent the form from submitting so that you can send the data with `fetch`.

The concept is more or less the same in React, we have to call `event.preventDefault()` on the `submit` event:

```
    function handleFormSubmit(event) {
        event.preventDefault();
    }

    return <form onSubmit={handleFormSubmit}>
        <input type="text" name="name" />
        <input type="submit" value="Add" />
    </form>;
}
```

Note that we preferred a named function definition over an inline function because eventually, we will have several lines of codes, not just the `event.preventDefault()`.

Recap

- You can make `<select />` and `<textarea />` controlled just like you would do for the `<input />`.
- Every input needs its own state variable.
- You can prevent the form from submitting with an `onSubmit` handler on the `<form />` that calls `event.preventDefault()`.

Was this helpful?

Chapter recap

Last updated June 2022

In this chapter, we learned how to work with basic forms; in the next chapter, we'll learn how to make our forms *accessible* and what that means.

If you're interested in more advanced forms, check out libraries such as **formik** and **react-hook-form**.

Recap

- `onChange` on an input will fire every time the user makes a change to that textbox.
- You can read the user's text with `event.target.value`.
- A controlled component is when you keep track of an **input**'s value as **state** and update it whenever it changes.
- To make an input controlled, you have to assign its value to a state variable and update that state variable whenever the input changes.
- Controlled component is a combination of `value={state}` and `onChange={event => setState(event.target.value)}`
- You can make `<select />` and `<textarea />` controlled just like you would do for the `<input />`.
- Every input needs its own state variable.
- You can prevent the form from submitting with an `onSubmit` handler on the `<form />` that calls `event.preventDefault()`.

Was this helpful?

What is accessibility?

Last updated June 2022

What is accessibility and why should you care?

Accessibility is the design and creation of web applications that can be used by **everyone**.
It's a practice that promotes **inclusion** because everyone (people with all abilities) will be able to use your website.

You may often see accessibility shortened as `a11y`, which means it's a word that starts with `a`, ends with `y`, and has 11 characters in between (`ccessibilit`).

We've already seen an example of accessibility where we recommended that you only use the `onClick` handler on `<button>` elements so that assistive technology software can understand through semantics (the meaning of the HTML code) that there is an action on this element when clicked.

Accessibility is mostly related to HTML, and since your React components are rendering HTML to the DOM in the end, you have to apply these accessibility best practices in React too.

Form-specific accessibility

onSubmit

The `onSubmit` event fires on a `form` element. If you expect users to fill and submit a form, then it's important to implement the `onSubmit` method rather than a `click` event on a button as the `submit` allows the user to control the form using their keyboard (by hitting `Enter` when they're done).

Labeling inputs

your form as the label makes your form accessible, for two main reasons:

1. Mouse users & users with a motor impairment will be able to click on the label tag to access the associated form control. This means there will be a larger clickable area for the input/textarea/select since clicking on the label or the control will activate the element.
2. Visually impaired users use screen-readers that read the associated label tag whenever a form element is focused. This allows them to hear the label that describes the input/textarea/select element.

What about the placeholder?

The `placeholder` attribute does **not** replace the `label` as its value gets hidden as soon as the user starts writing inside the element.

So how do we add labels to a form? We'll detail it in the next lesson :)

Recap

- Accessibility is the design and creation of web applications that can be used by **everyone**.
- Accessibility is a practice that promotes **inclusion** because everyone (people with all abilities) will be able to use your website.
- Expecting users to submit a form? Implement the `onSubmit` on the form instead of a `click` on a button.
- Add `label` to every input/textarea/select element.

Was this helpful?

Labeling form items

Last updated January 2023

Here's how you can add a `label` to an `input` in **HTML** (it will be slightly different in React):

```
<form>
    <label for="login-email">Email: </label>
    <input type="email" id="login-email"
placeholder="alex@email.com">

    <label for="login-password">Password: </label>
    <input type="password" id="login-password"
placeholder="Password">

    <input type="submit" >
</form>
```

The `<label>` element needs a `for` attribute pointing to the `id` of the input/select/textarea element it's pointing to.
Notice how we made it work by adding an: `id="login-email"` and then referring to it with `for="login-email"`. And for the password field: `id="login-password"` and then referring to it with `for="login-password"`.

Ensure that the `id` is unique as that `id` should only be used once per HTML page.

Labels in React

Labels in React are slightly different for the same reason why `class` is different in React.
Remember that `class` has to be used as `className`. React is passing all these props down to the DOM, which expects them to be valid properties of an HTML element.

So here's the same form but in JSX:

```
<form>
    <label htmlFor="login-email">Email: </label>
    <input type="email" id="login-email"
placeholder="alex@email.com" />

    <label htmlFor="login-password">Password: </label>
    <input type="password" id="login-password"
placeholder="Password" />

    <input type="submit" />
</form>
```

Recap

- A `<label />` needs an `htmlFor` attribute that points to the `id` of the element.

Was this helpful?

Chapter recap

Last updated June 2022

Nice work!

It's time for a project. See you there!

Chapter recap

- Accessibility is the design and creation of web applications that can be used by **everyone**.
- Accessibility is a practice that promotes **inclusion** because everyone (people with all abilities) will be able to use your website.
- Expecting users to submit a form? Implement the `onSubmit` on the form instead of a `click` on a button.
- Add `label` to every input/textarea/select element.
- A `<label />` needs an `htmlFor` attribute that points to the `id` of the element.

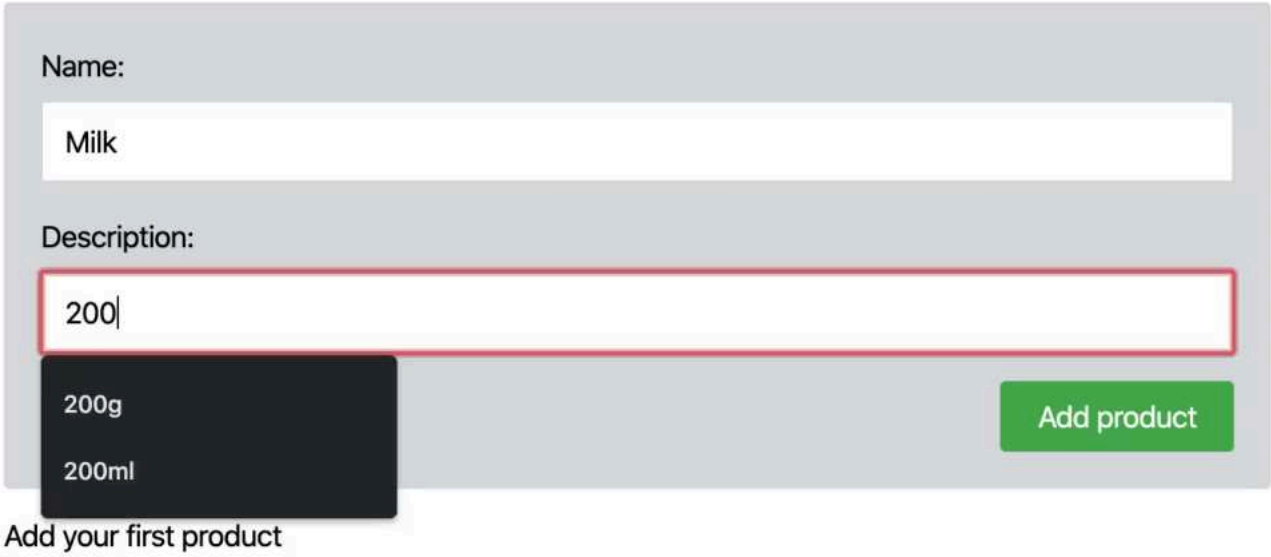
Was this helpful?

Project IV

Last updated September 2022

Project IV aims to recap the knowledge you've acquired so far in this course, with a focus on:

- Multiple states
- Immutability
- (Accessible) forms



Arrays of objects & immutability

We've seen how to immutably modify arrays of primitives (such as strings, numbers, booleans), but what about *arrays of objects*?

There's a common caveat when it comes to arrays of objects, which relates to the concept of shallow and deep cloning,

```
const users = [{
   id: 1,
   name: "Alex"
}];

// create a (shallow) copy
const usersCopy = [ ... users];
usersCopy[0].name = "Sam";
console.log(usersCopy[0].name); // "Sam"
console.log(users[0].name); // Also changed! "Sam"
```

Notice how the object inside the array was changed even though we created a copy of it using the `[ ... users]`.
That's because the spread syntax creates a *shallow* copy rather than a *deep* copy.

A shallow copy means that it creates a new array, but the objects are still referring to the old ones. Hence changing an object in the new array will also change in the old one because they're pointing to the same object.

A deep copy means that the new array contains new objects, so changes to those new objects will not affect the old ones in the original array.

Deep clone in JavaScript

Browsers have recently shipped a function called `structuredClone` that allows you to create a *deep clone* in JavaScript without having to resort to an external library (such as `lodash`).

However, keep in mind that creating a *deep clone* on every state update is an expensive operation, so we'd like to avoid creating deep clones.

The solution

as when working with arrays of objects because it is creating a new copy of the array, which is enough to notify React that the state has changed:

```
const users = [{
   id: 1,
   name: "Alex"
}];

// create a (shallow) copy
const usersCopy = [ ... users];

// are they the same? If `false` then React thinks
// that the state has changed
console.log(users === usersCopy); //false
```

So just because we created a new copy of the users, this is going to make React think that we've changed our state, which is great for us as we can go ahead and update the `usersCopy` even though this will end up updating the previous objects.

Note that there are libraries that can make the process of updating arrays of objects easier, such as **immer** (note: `immer` is used in Redux Toolkit, Chapter 90 of this course), **immutable-js**, and others.

However, we won't be using them for now.

Actions inside .map

Here's a tip that may come in handy.

Let's say you have a list of items you're iterating over in JSX with `.map`, and you want to have a `button` for every one of these items, and that button should `log` the item that was clicked.

```
function List() {
    const items = [1, 2, 3];
```

```
    }

    return <ul>
        {items.map(item => <li key={item}>
            {item}
            <button onClick={handleButtonClick}>Log me</button>
        </li>)}
    </ul>
}
```

Notice here how we cannot know which button was clicked because we need to pass the `item` to the `handleButtonClick`.

Here's how you can pass the current item details to the event handler:

```
function List() {
    const items = [1, 2, 3];

    function handleButtonClick(item) {
        console.log(item);
    }

    return <ul>
        {items.map(item => <li key={item}>
            {item}
            <button onClick={() => handleButtonClick(item)}>Log
me</button>
        </li>)}
    </ul>
}
```

Remember that the `onClick` should **always** receive a function definition, not a function call, hence why it has to start here with `() =>` and we give it `item` as a parameter.

Recap

- The spread syntax creates a *shallow* copy rather than a *deep* copy.

- A deep copy creates a new array with new objects.
- Creating a new copy of an array is enough for React to update the state because the array itself has changed (even if the objects inside are referring to the old ones).
- `onClick=() => handleButtonClick(item)` allows you to pass the `item` as a parameter (assuming `item` is a variable in the current scope).

Was this helpful?

Project recap

Last updated August 2023

Congratulations on completing Project IV 🎊

Taking a step back

Let's take a step back and see what we have learned so far.

- ReactDOM
- JSX
- Components
- Props
- State
- Hooks (only the `useState` hook for now)
- Events
- Forms
- All sorts of state updates (with immutability)

Limitations of this project

You may have noticed two main limitations of this project:

1. When using the keyboard and submitting the form, there's no proper way to programmatically move the `focus` (the blinking cursor) from the `description` field back to the `name` field. We will need `refs` for that.
2. The `StoreFront.js` file became quite large. It would be ideal if we could refactor its logic into two main components:

```
import React from "react";

function StoreFront() {
```

```
        <ProductsList />
    </>;
}
```

But we can't **yet**.

That's because both of these components depend on the same `state`.

This is why we will learn about **lifting state up** in the following few chapters, which allows us to split components into multiple smaller ones.

But first, let's take a closer look at the `create-react-app` and Vite!

Was this helpful?

create-react-app & Vite

Last updated December 2023

Create React App (or CRA) *was* an officially supported way of creating single-page React Apps. However, this project stalled and recently became outdated. A more modern alternative is Vite.

Vite's React template

Vitejs.dev recently emerged as an excellent alternative to CRA. We're covering both options in this course. We do recommend *vite* as it's more modern and way faster.

The instructions for vite are provided at the end of this lesson.

You're encouraged to start using Vite (or create-react-app, if you wish) in parallel with this course so that you can practice writing React apps outside of this course.

The future of Vite and create-react-app

If you visit the official react docs on **starting a new react project**, you will notice that they recommend you use a *production-grade React framework*. CRA is no longer mentioned and Vite is hidden under the *Deep dive > Can I use React without a framework?* section.

Since React is only a library that takes care of the UI, it wanted to recommend React frameworks instead, such as Next, Remix, and Gatsby. However, please note that it is essential to learn how to use React as a library first before using React as part of a framework. It's like learning how the gearbox works before learning how to drive a car. So, against the React recommendation, we actually recommend you use Vite in parallel with this course and for your first project. Your first project won't be perfect, it's a learning experience.

a full-fledged framework. Keep in mind that learning a framework requires learning new concepts. This would be useful if these concepts are relevant to your website (for example, server-side rendering, storage, data fetching, etc.).

Installation

To get started with Vite, you need to ensure that **NodeJS** is installed on your computer. Then open a terminal and run the following:

```
npm create vite@latest
```

This will open up an interactive questionnaire where you have to choose `React` and then the `JavaScript` variant.

In the next lesson, we'll explore the Scripts/commands we get out of the box with Vite.

Recap

- Create React App (or CRA) was an officially supported way of creating single-page React Apps. However, it's currently discontinued.
- Vite (vitejs.dev) recently emerged as an excellent alternative that is more modern and faster than CRA.
- While the official react docs don't recommend using Vite for production applications, we still recommend it **for learning purposes**. Make sure to use it before choosing a more advanced option.
- Get started with Vite by running: `npm create vite@latest`

Was this helpful?

Scripts

Last updated October 2023

Now that you've run the above command, `cd` (change directory) into the new directory that was created and then run `npm install` to install all the dependencies.

We'll cover the folder structure in the next lesson, but for now, let's open the generated `package.json` file and look at the `scripts`:

```
{
    "scripts": {
        "dev": "vite",
        "build": "vite build",
        "lint": "eslint src ...",
        "preview": "vite preview"
    }
}
```

You can use these commands, for example, running `npm run dev` and `npm run build`.

npm run dev

This is the main command you will use whenever you want to start working on your app.
`npm run dev` will start a development web server, which you can access by going to [localhost:5173](). However, the port could be different so please check your terminal.

This command will use the **development** version of React, which will be covered at the end of this chapter.

npm run build

create an optimized version of your app.

It will minify all your scripts and perform several optimizations before deploying your app. The optimized files will be placed in a new folder called `dist`.

We will cover deployment in this course in a later chapter.

This command will use the **production** version of React, which will be covered at the end of this chapter.

npm run preview

This command allows you to preview a *production* build on your machine. This is often helpful to test how a production build behaves without deploying your code.

npm run lint

The lint command will check your code quality for common mistakes and potential improvements.

Recap

- `npm run dev` will start the development server. It uses the development version of React.
- `npm run build` will build your application for deployment. It uses the production version of React.
- `npm run lint` will check your code quality for common mistakes and potential improvements.

Was this helpful?

File Structure

Last updated October 2023

Out of the box, Vite generates the following folder structure:

```
.
├── src
└── public
```

The `src/` folder contains the `<App />` component, where you should put all your scripts & styles.
The `public/` folder is where you should put all your images and fonts (if any).

The `index.html` file is placed at the root of the folder created by Vite.

The <App /> component

Your application's starting point will be the `src/main.jsx` file. This file imports the App component from `App.jsx`.

The `main.jsx` file contains the following code:

```
ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
)
```

From the code above, we can guess that the `index.html` contains a `<div id="root"></div>`, which you can verify by checking out the `index.html` file. It also enables React's strict mode. This is introduced in Chapter 37.

Choosing an appropriate folder structure is a heavily debated subject, and that's natural because it comes down to several factors such as personal preferences, the complexity of the project as well as the type of project.

Note that if you (or your company) already prefer a certain folder structure, then go ahead and use it.

To better answer this question, we'd like to split the answer into four categories based on the project size.

Micro-sized projects

This refers to projects that have less than approximately five components.
In these cases, you do not need to create any folders.
You can create new components in `src/`; for example, your `src/` would look like this:

```
.
└── src
    ├── App.js
    ├── Navbar.js
    └── Footer.js
```

It's not a bad idea to start with this folder structure and then refactor them into folders once your project gets bigger.

Small-sized projects

Suppose you're working on a small-sized project, which is to say, a project that contains less than approximately 20 components. In that case, you can choose to group by feature, for example:

```
.
└── src
    ├── products
    │   ├── Product.js
```

```
│   ├── store
│   │   ├── StoreFront.js
│   │   └── Checkout.js
│   └── user
│       ├── User.js
│       └── Profile.js
```

Medium-sized projects

With medium-sized projects, you will most likely have growing complexity outside the `components` as well; for example, you may need several `helper` files, several `classes` files, etc.

In that case, it makes sense to follow the recommendation from Small-sized projects and move all the components into a `components/` folder. This will make room for other folders such as `helpers/`, or `classes/`. For example:

```
.
└── src
    ├── components
    │   ├── products
    │   │   ├── Product.js
    │   │   ├── ProductDetails.js
    │   │   └── ProductQuantity.js
    │   └── store
    │       ├── StoreFront.js
    │       └── Checkout.js
    ├── helpers
    │   ├── date.js
    │   └── currency.js
    └── classes
        └── backend.js
```

Large-sized projects

Large projects will most likely have specific requirements; thus, it will be up to them to decide on a folder structure.

Note that the above folder structure works for large-sized projects. But for example, your large-sized project may be serving mostly different content for `admin/` and `client/`; in that case, it may make sense to split the components folder into two folders, 1 for **admin** and one for **client**.

We can't tell for sure as it really depends on the number of similarities and differences between these two.

General recommendations

Here are two general recommendations:

1. Avoid too much nesting as it becomes tougher to refactor and import files.
2. Don't spend too much time deciding on a file structure. You can always start gradually and refactor as needed.

Recap

- The starting point of your application will be the `src/main.jsx` file and then the `src/App.jsx` file.

Was this helpful?

React development vs. production

Last updated May 2023

React is distributed in 2 versions: **React development** and **React production**.

React development

React development is used when you run `npm run dev`.
The development version is not minified. It contains all the comments from the React source code and also includes development helpers such as *warnings* that will show up in some scenarios.
In fact, you probably have seen those warnings already in this course that show up in the console.

The development version should **only** be used in development and never in production, as it ends up slowing down your app.

React production

React production is used when you run `npm run build`.
The production version is minified, strips out all the comments, and does not include development helpers and warnings.

This is why it's suitable for production.
The production version should be used whenever you deploy your website.
If you're using create-react-app or vite, then you don't need to worry about this, as the versions are automatically swapped out for you when you run `npm run build`.

Difference in sizes

between development and production.

- react.development.js (**111KB**) vs react.production.js (**11KB**)
- react-dom.development (**1.1MB**) vs react-dom.production.min.js (**132KB**)

Note that during this course, we are using the Development version of React (with some modifications) as we'd like you to see the warnings that you get while developing on your computer ;)

Recap

- React development is used when you run `npm run dev`.
- The development version is not minified, contains all the comments from the React source code, and also includes development helpers
- React production is used when you run `npm run build`.
- The production version is minified, strips out all the comments, and does not include development helpers and warnings.

Was this helpful?

Chapter recap

Last updated May 2023

Nice work!

Chapter recap

- Create React App (or CRA) was an officially supported way of creating single-page React Apps. However, it's currently discontinued.
- Vite (vitejs.dev) recently emerged as an excellent alternative that is more modern and faster than CRA.
- While the official react docs don't recommend using Vite for production applications, we still recommend it **for learning purposes**. Make sure to use it before choosing a more advanced option.
- Get started with Vite by running: `npm create vite@latest`
- `npm run dev` will start the development server. It uses the development version of React.
- `npm run build` will build your application for deployment. It uses the production version of React.
- `npm run lint` will check your code quality for common mistakes and potential improvements.
- The starting point of your application will be the `src/main.jsx` file and then the `src/App.jsx` file.
- React development is used when you run `npm run dev`.
- React production is used when you run `npm run build`.

Stateless vs. Stateful components

Last updated July 2022

React Components can either be *stateless* or *stateful*.
The distinction is with regard to whether the component handles state or not.

Stateless components do NOT manage state internally. So you will not find `useState` calls inside of them. On the other hand, *stateful* components will manage at least one state variable.
We've already had several examples of stateful components, as well as stateless components.
The only reason we're making this definition now is that as your application grows, you will end up splitting one big stateful component into several smaller components. Some of these components will be stateless.

This will make it easier for us to refactor and simplify our application as it grows in size.
If this sounds too theoretical, don't worry because it is theoretical. We will be breaking down this concept step by step in the following few chapters.

Stateless components can be interactive

Even though stateless components do not manage state internally, they can still be interactive.
For example, a stateless component can contain a `form` as well as a textbox and a submit button. However, the state will be managed by its parent component.
This is the first concept that we will be covering in this chapter.

To make that possible, we first must realize that `props` can also pass functions. In the next lesson, we'll explain what that means :)

Recap

- *Stateful* components manage state internally (At least one call to `useState` inside the component).
- Nevertheless, stateless components can be interactive. However, the state will be managed by a parent component.

Was this helpful?

Passing function props

Last updated July 2022

We've seen examples of props so far that contained booleans, strings, numbers, arrays & objects.
However, props can contain functions too. Here's an example (for convenience, both components are defined in the same file):

```
function App() {
    function handleWelcome() {
        console.log("Hello World");
    }

    return <StoreFront onWelcome={handleWelcome} />;
}

function StoreFront(props) {
    props.onWelcome();

    return <div>Store renders here</div>;
}
```

Notice what's going on here?

1. We defined a `handleWelcome` method in the `App` component which is the parent component.
2. Then we rendered the `StoreFront` component and passed a prop called `onWelcome`.
3. This `onWelcome` prop is a function (a reference to the function `handleWelcome`)
4. From within the `StoreFront` component, we can call that function with `props.onWelcome()`.

We will see `Hello World` printed on the console.

focus on how to do it, and the "why" will become clearer in the next few steps.

Naming convention

When passing functions, follow the below naming conventions:

1. Functions are still called using our `handleSubjectEvent`. (The above example is simplified, it does not have an `Event` thus why we omitted it)
2. For props that are *functions*, start them with `onSubjectEvent`. This makes it easier to distinguish prop items that are functions.

Here are more examples:

```
function App() {
    function handleStoreOpen() {

    }

    function handleItemClick() {

    }

    return <StoreFront
            onStoreOpen={handleStoreOpen}
            onItemClick={handleItemClick}
        />
}
```

Notice how `onStoreOpen` is a prop passing a function and the same for `onItemClick`.

Recap

- Props can contain functions.
- A component can pass a function to a child component as a prop.

Was this helpful?

Chapter recap

Last updated July 2022

Now that we know how to pass functions, we'll use that knowledge to lift the state up, which we'll cover in detail in the next chapter.

Recap

- *Stateless* components do NOT manage state internally. (No calls to `useState` inside the component)
- *Stateful* components manage state internally (At least one call to `useState` inside the component).
- Nevertheless, stateless components can be interactive. However, the state will be managed by a parent component.
- Props can contain functions.
- A component can pass a function to a child component as a prop.
- Naming convention: `onSubjectEvent={handleSubjectEvent}`

Was this helpful?

Lifting Form's state up

Last updated July 2022

Now that we know how we can pass functions as props in React, let's take a look at this React component:

```
// index.js
import {useState} from "react";

function App() {
    const [name, setName] = useState("");

    function handleNameChange(event) {
        setName(event.target.value);
    }

    return <div>
        <h2>Hello {name}</h2>
        <form>
            <label htmlFor="name">Name: </label>
            <input type="text" id="name" value={name} onChange=
{handleNameChange} />
        </form>
    </div>
}
```

We would like to refactor it into the following:

```
// index.js
import {useState} from "react";
import NameForm from "./NameForm.js";

function App() {
    const [name, setName] = useState("");

    function handleNameChange(event) {
```

```
    return <div>
        <h2>Hello {name}</h2>
        <NameForm />
    </div>
}
```

Notice how we refactored the `<form>` into a `NameForm` component.
Also, note that this won't work as we'll need to pass the `name` and `onChange` events to the `NameForm` component. Keep on reading to see how we can do that.

1. Pass the "value" and "onChange"

The `NameForm` component needs to show a textbox and update its value whenever it changes; thus, it needs to set the `value` and the `onChange` props.
For that, we start by passing them down from the App component:

```
// index.js
import NameForm from "./NameForm.js";

function App() {
    const [name, setName] = useState("");

    function handleNameChange(event) {
        setName(event.target.value);
    }

    return <div>
        <h2>Hello {name}</h2>
        <NameForm name={name} onNameChange={handleNameChange} />
    </div>
}
```

A couple of things to note:

- `name={name}` is passing down the state variable name

handleNameChange (what we saw in the last chapter).

- onNameChange follows the same naming convention presented in the last chapter.
- Notice how the state is created and maintained in the parent component.
- The App component is a *stateful* component because it manages the state.

2. Use the "value" and "onChange"

Now in the NameForm component, we can use these 2 props:

```
//NameForm.js
export default function NameForm(props) {

    return <form>
        <label htmlFor="name">Name: </label>
        <input type="text" id="name" value={props.name} onChange=
{props.onNameChange} />
    </form>
}
```

A couple of things to note:

- This is a *stateless* component as it does NOT manage state. Even though there is a textbox, this component has an onChange handler that calls props.onNameChange, which belongs to its parent component.
- The parent component handles the state.
- Notice how the value={name} changed to value={props.name} because the name is not an internal state anymore but rather a prop received from the parent.
- The same thing applies to the onChange.

In conclusion

Do you see how the App is *stateful*, and the NameForm is *stateless*?
So the state is only managed in the parent component App.

called lifting state up, but don't worry about that for now.)

Recap

- When refactoring a form into a child component, the form will most likely become stateless, and the state will become managed by the parent.
- To be able to do that, you have to pass the `value` and `onChange` from the parent down to the child and then use them as `props.value` and `props.onChange`.

Was this helpful?

Lifting state up

Last updated July 2022

In the previous lesson, we showed how state could be defined in one component and used in the child component.
In this lesson, we'll explore the concept of *lifting state up*.

To share a state between two components, you can move the state **up** to their closest common ancestor.
This means, if you have two components that depend on the same state, then the state will be defined in their closest parent component.

Let's say we'd like to build a Todo List; we'll need a `form` to add the "todo" as well as a `ul` to list those todos.

```
import {useState} from "react";

function TodoApp() {
    const [todos, setTodos] = useState([]);
    const [entry, setEntry] = useState("");

    function handleEntryChange(event) {
        setEntry(event.target.value);
    }

    function handleFormSubmit(event) {
        event.preventDefault();
        // add the new entry
        setTodos([ ... todos, entry]);
        // reset/empty the textbox
        setEntry("");
    }

    return <>
        <form onSubmit={handleFormSubmit}>
            <label htmlFor="todo">Enter To do: </label>
```

```
        </form>
        <ul>
            {todos.map((todo, index) => <li key={index}>{todo}
</li>)}
        </ul>
    </>;
}
```

Now let's refactor this component into 2 additional components:

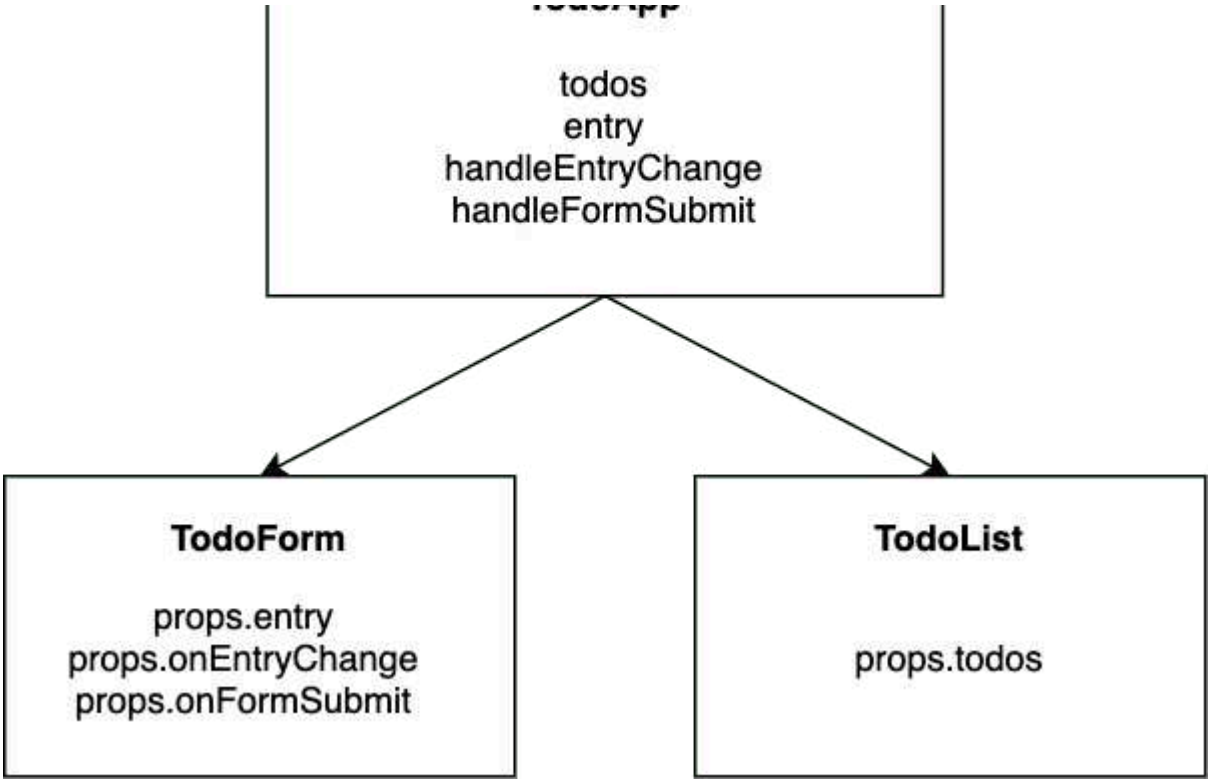
- `<TodoForm />` containing the `<form>` and its children
- `<TodoList />` containing the `<ul>` and its children

The problem is that both components, `TodoForm` and `TodoList` depend on the same states, which are `todos` and `entry`.

The form needs the `todos` state to be able to add the new `entry` to the todos array.

Thus, we have to lift the state up and make that state definition in the parent component, which is `<TodoApp />`.
Then this TodoApp component will pass the state and the onChange methods down to both components.

Here's a drawing of what it will look like:



So the state gets defined in the parent component.

Here's how it looks like when refactored:

```
// TodoApp.js
import {useState} from "react";
import TodoForm from "./TodoForm.js";
import TodoList from "./TodoList.js";

function TodoApp() {
    const [todos, setTodos] = useState([]);
    const [entry, setEntry] = useState("");

    function handleEntryChange(event) {
        setEntry(event.target.value);
    }

    function handleFormSubmit(event) {
        event.preventDefault();
        setTodos([ ... todos, entry]);
        setEntry("");
    }
```

```
        <TodoForm entry={entry} onEntryChange={handleEntryChange}
onFormSubmit={handleFormSubmit} />
        <TodoList todos={todos} />
    </>;
}


// TodoForm.js
function TodoForm(props) {
    return <form onSubmit={props.onFormSubmit}>
        <label htmlFor="todo">Enter To do: </label>
        <input type="text" id="todo" value={props.entry}
onChange={props.onEntryChange} />
    </form>;
}


// TodoList.js
function TodoList(props) {
    return <ul>
        {props.todos.map((todo, index) => <li key={index}>{todo}
</li>)}
    </ul>;
}
```

Note that it is technically possible to have the `entry` state inside the `TodoForm` component. When it comes to lifting state up, it's almost impossible to give a recommendation that works for 100% of the scenarios.

However, having all the shared state in the closest common ancestor component has some benefits (even though the code might be a bit more lengthy). Here are those benefits:

- The common ancestor becomes the **single source of truth**. This makes it substantially easier to find bugs and fix them, as you know that there's only one place where this state will be changed.
- Having this centralized location for the state, makes it easier to maintain consistency in your application. If you create some validation logic for the `entry`,

Recap

- To share a state between two components, you can move the shared state up to their closest common ancestor.

Was this helpful?

Chapter recap

Last updated July 2022

We lift the state up because we want the data to flow down.

When the data flows down, your app becomes more predictable, and finding and isolating bugs becomes easier.

The naming convention of `onSubjectEvent` is important as it serves as a reminder that this is an **event**.

Remember that there are several ways to solve the same challenge in React. React itself tries not to force you into a single way. Throughout this course, we try to give you recommendations because when starting with a new framework, it can be hard for you to make your own decisions. While working with larger companies or on larger projects, you may notice that different approaches are being taken, which is perfectly fine.

Chapter recap

- When refactoring a form into a child component, the form will most likely become stateless, and the state will become managed by the parent.
- To be able to do that, you have to pass the `value` and `onChange` from the parent down to the child and then use them as `props.value` and `props.onChange`.
- To share a state between two components, you can move the state up to their closest common ancestor.

Was this helpful?

Breaking up components

Last updated April 2023

While building an app with React, you may end up with big components containing several hundreds of code lines.
The components need to be refactored and broken up into several smaller components.

Breaking up a component into a smaller one has several advantages, mainly:

1. Reusability: You can reuse the smaller components anywhere in your application if you need to.
2. Isolate bugs: If you found a bug with a part of your UI, you will be able to isolate it to 1 or 2 small components rather than one huge component.

This process is called **abstraction** and is not different than what we've seen in the previous chapter; for example:

```
import React from "react";

function App() {
    return <>
        <form>{/* form to add a product */}</form>
        <ul>{/* list of products */}</ul>
    </>
}
```

The above pseudo-code can be refactored into:

```
import React from "react";

function App() {
    return <>
        <AddProductForm />
        <ProductsList />
```

We have abstracted the form, which is most likely going to be around ten lines of code, into the `<AddProductForm>` component. We also abstracted the list of products in the `<ul>`, which will most likely be around five lines of code into the `<ProductsList />` component.

This makes it easier to understand what our app is doing because the component names are descriptive.
It also lets us reuse these components elsewhere in the application.

You can think of abstraction as the process of creating a component out of some part of your application.

Having said that, do not rush and create a component out of everything. We have guidelines below that help you decide when to break up a component into smaller ones.

When to break up a component?

Breaking up a component is not an exact science. The advice you will see below is slightly opinionated and is not a set of fixed rules but rather some recommendations. There is not a single correct answer.

We recommend that you follow the **Work In Progress** approach where:

1. You start building a component
2. then you realize it needs to be refactored
3. You refactor it
4. Assess if it's good enough and continue working

In an ideal world, you would have a clear map (or tree) of all the React components that you will have in your application; however, this is far from reality and almost impossible (due to changing business requirements). This is why we recommend you think about your components' structure (but not overthink it) and constantly refactor as you need to.

Here are the factors that you should consider to break up a component:

1. Need for reusability

When you notice a part of your UI (a few React elements in your JSX, for example, a `ul` and its children) needs to be reused somewhere else, you can create a component out of that piece of the UI.
For example, when building a supermarket app, you will most likely need to show the `Product` in several different views; this should encourage you to create a `<Product />` component so that you can reuse it.

2. The component becomes "too complicated"

When your component becomes too complicated, it is a good idea to break it into smaller components.
There is no specific threshold as to when a component becomes "too complicated"; it's also a subjective classification.
Some factors that will make you realize that a component is too complicated are:

- Lines of code (For example, more than 200 lines of code - different teams will have different opinions regarding this)
- Hard to follow: it's not clear what the component is doing, and it takes a lot of time to understand how it works
- Performance problems: The component has become too slow when re-rendering

3. Expecting further development

When you are most likely sure that this part of the UI will become more complicated over time, then it's a good idea to break it up. For example, your application has a list of *orders* that currently show the order id and order name. However, you know for sure that the team is planning on adding further development to that list, for example, the ability to track/cancel/edit the order.

application can become more complicated over time, this is why it's important to only refactor a certain part of the UI if you're 99% sure that it will be further developed.

When refactoring, you will most likely be creating stateless components for some part of the UI that does not depend on state at all, and this is perfectly fine.
For example, if you have a static footer, you should still make a component out of that footer.

You will also be using your knowledge of lifting state up, which you've already seen in the previous chapter. We'll be further practicing in the following few challenges.

In the next lesson, we'll take a look at an example where we refactor an existing app.

Recap

- Refactoring components promote reusability and help in bug isolation.
- Follow the **Work In Progress** approach when building your application.
- Factors for breaking up a component into smaller ones:
- 1. Need for reusability
- 2. The component becomes "too complicated"
- 3. Expecting further development

Was this helpful?

Room Booking App

Last updated July 2022

Let's take an example of a *Room Booking App*, with the following business requirements:

- List the rooms
- You can click on a room to select it and then book it by clicking on the **Book** button
- Right now, we only have the location of the room, but we're working on adding support for images, descriptions, and more features.

With the above requirements, we would most likely come up with the following structure:

```
.
└── <RoomBooking />
    ├── <RoomList />
    │   ├── <Room />
    │   ├── <Room />
    │   └── <Room />
    └── <BookRoom />
```

Note that even though we only have the *location* of the room, we decided to create a `<Room />` component because the business requirements are quite confident that there will be further product development. In case you did not realize that from the start, no big deal as you can refactor it later on.

The above components may change drastically if you have more information; this is why it's important to always think about this as **Work In Progress**.
Leave some time to be able to refactor as new business requirements come on.

Recap

- Keep in mind that they will change in the future. Stay in a **Work In Progress** mentality. Leave time to refactor when necessary.

Was this helpful?

Chapter recap

Last updated July 2022

Nice!

Recap

- Refactoring components promote reusability and help in bug isolation.
- Follow the **Work In Progress** approach when building your application.
- Factors for breaking up a component into smaller ones:
- 1. Need for reusability
- 2. The component becomes "too complicated".
- 3. Expecting further development
- Try to sketch the components you'll be creating based on the current business requirements
- Keep in mind that they will change in the future. Stay in a **Work In Progress** mentality. Leave time to refactor when necessary.

Was this helpful?

Project V

Last updated July 2022

Awesome that you made it so far!
This project aims to refactor the end result from Project IV.

In **Project IV's recap**, we denoted two limitations; we will tackle the second limitation in this project. Which is:

The StoreFront.js file became quite big. It would be ideal if we could refactor its logic into two main components:

```
function StoreFront() {
    return <>
        <AddProductForm />
        <ProductsList />
    </>;
}
```

Let's go ahead and refactor the StoreFront component!

Was this helpful?

Project recap

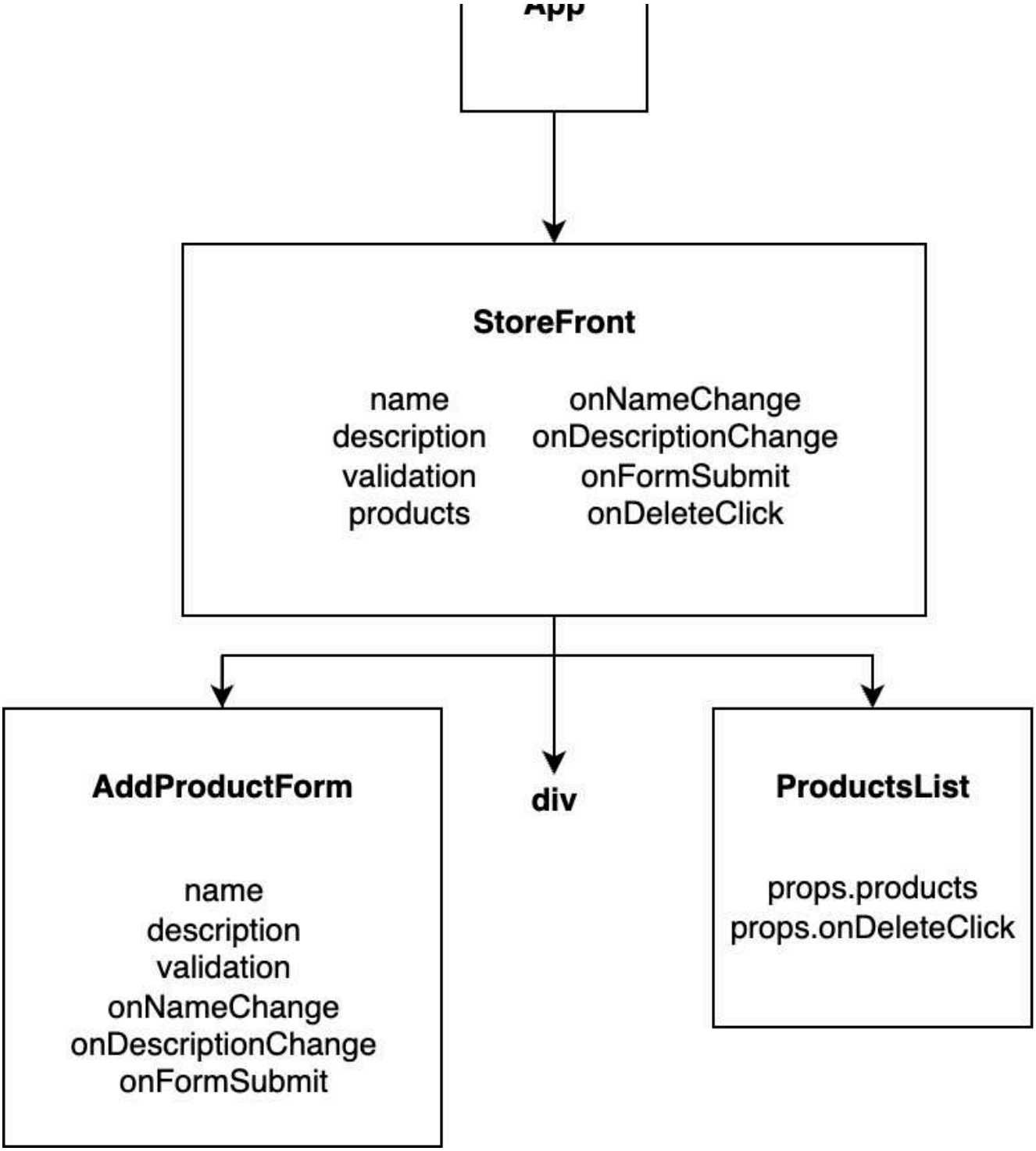
Last updated August 2023

Congratulations on completing project V! 🎊

Here's what the solution for Project V looks like; you've got the App component, which has the child: `StoreFront`.
The `StoreFront` component holds the state for `name`, `description`, `validation`, and `products` and defines 4 methods: `onNameChange`, `onDescriptionChange`, `onFormSubmit` and `onDeleteClick`.
The `StoreFront` component renders 3 elements:

1. The `AddProductForm`
2. A div element (conditionally: it shows `<p>Add your first product</p>` based on the condition of `products.length === 0`)
3. The `ProductsList`

The `AddProductForm` and `ProductsList`, receive the necessary variables/methods as props.



An important thing to visualize here is how the `state` has been lifted to the **most common ancestor**, the `StoreFront` component.

We didn't define the state in the App component because you should only lift it up to the most common ancestor.

Because we need those state variables in `AddProductForm` and `ProductsList`, then the most common ancestor is `StoreFront`.

Remember that lifting the state up has the following two benefits:

- The common ancestor becomes the single source of truth. This makes it substantially easier to find bugs and fix them, as you know that there's only one place where those state variables can be changed.
- Having this centralized location for the state makes it easier to maintain consistency in your application. If you create some validation logic for your state, you will only have to do so once in the common ancestor.

There are several ways you can solve this challenge, and for now, we do prefer the method explained and given in the official solution due to the benefits above.

However, keep in mind that forms in React require some boilerplate code, but we can trim that down using a form helper library (for example, using **react-hook-form**).

Taking a step back

Let's take a step back and see what we've accomplished so far.

You're most likely enjoying the journey and forgot that you've already learned all of these React topics:

- Pure functions
- Closures
- Events
- JSX Advanced
- Advanced Array destructuring
- Improved UI Kit
- Hooks (just one for now)
- State
- Conditional state changes
- Conditional rendering
- Immutability
- Array immutability

- Object Immutability
- State with Objects
- Forms
- Accessibility
- Advanced State
- DOM Events
- Folder structure
- Custom Events
- Lifting state up
- Virtual DOM

That's quite a lot! So take a minute to enjoy that :)

Was this helpful?

React Dev Tools

Last updated May 2023

React Dev Tools is a developer tool you can install to your browser. It helps you understand and debug your React app.

The extension is available for Chrome & Firefox:

- React Dev Tools for Chrome
- React Dev Tools for Firefox
- React Dev Tools for Edge

Go ahead and install it for your browser of choice. Unfortunately, it's currently not available for Safari.

Note that it will only activate when you're browsing a website that is using React.

The extension will come in handy once your application grows bigger as it has many convenient features. Let's take a look at those features.

React development/production

When you're on a website that uses React, the extension will either show you this message:

```
This page is using the production build of React. ✅
Open the developer tools, and "Components" and "Profiler" tabs
will appear to the right.
```

or this one:

```
This page is using the development build of React. 🚧

Note that the development build is not suitable for production.
Make sure to use the production build before deployment.
```

```
will appear to the right.
```

This would serve as a good reminder if you did not use the production version for React when deploying your site. A reminder that if you're using create-react-app, then you do not have to worry about this.

Also, note that on [React Tutorial](), we use the **development** version because we'd like to show you the warnings from the React library as you learn React. So this is an exception to the rule. Normally all production websites should use the production version.

Browse Components

When you're on a website that's using React, right-click anywhere on the page, and select **Inspect Element** or **Inspect** and then browse to the **Components** tab, which you should find at the end of the tabs.

Check out how we can use the React Dev Tools to inspect and edit our state as well as to go from a React component to its corresponding DOM element:

This behavior can be modified by clicking on the **gear** icon at the top right of the extension, then **Components** tab, then toggling the first rule under **Hide components where**.

Official React Dev Tools tutorial

If you want to dig deeper into the React Dev Tools, check out this excellent **tutorial**.

Recap

- React Dev tools makes it easier to debug your React apps.
- The extension only activates on sites that are using React.
- It tells you whether you're using the development or production version of React.
- You can browse components, inspect & edit state within your app.

Was this helpful?

Profiler

Last updated July 2022

React Dev Tools includes another tab which is called **Profiler**.

This tab is useful when there are a lot of components in your app, as it lets you see which components are re-rendered and why. When improving the rendering performance, you will often realize that there was a component that did not need to re-render.
We will have an entire chapter about that later in the course.

Feel free to try this tab by clicking the record button and then interacting with your app (add/remove items, click on elements that have an `onClick` handler). However, we only have five components with the current project that we built, so we will not see any value with the profiler yet.

Recap

- The profiler tab in React Dev Tools will help us spot rendering performance issues in our React apps.

Was this helpful?

Chapter recap

Last updated May 2023

Now that you know about React Dev Tools keep it in mind next time you're stuck on a bug. Try to use it and see if it helps you find the bug ;)

Recap

- React Dev Tools makes it easier to debug your React apps.
- The extension only activates on sites that are using React.
- It tells you whether you're using the development or production version of React.
- You can browse components, inspect, and edit state within your app.
- The profiler tab in React Dev Tools will help us spot rendering performance issues in our React apps.

Was this helpful?

State updates are asynchronous

Last updated March 2023

State updates in React are *asynchronous* which means that the state does not necessarily update **instantly**.

This is by design to improve the performance of React applications.

When you update the state in React, this will require *re-rendering* of your component (and potentially other components), which could be an expensive operation. This is why React batches several state updates together and combines them into one re-rendering in order to make your app more responsive and reduce the amount of work the browser has to do.

So let's take this example:

```
import React, {useState} from "react";

function App() {
    const [date, setDate] = useState(new Date());
    const [counter, setCounter] = useState(0);

    console.log("rendered"); //allows us to visualize re-renders

    function handleButtonClick() {
        setDate(new Date());
        setCounter(counter + 1);
    }

    return <button onClick={handleButtonClick}>Click me</button>
}
```

The `setDate()` is setting the current date by calling **new Date()**.

Now answer these questions:

2. How many re-renders will this component perform when clicking the button?

Here are the answers:

The answer to the first question is: there are two state updates. However, the component will only re-render once.

That's because React batches (merges) these two state changes and performs them at the same time. This makes your app respond faster to user interactions.

You should not know the specifics of how/when this batching occurs, as it's a React feature. However, this batching can create a caveat in your application as it grows more complicated. And that's what we're going to cover in the next lesson.

- ✔ As of React 18, React now batches state updates even if they were inside an event handler or a promise callback.

Recap

- State updates in React are asynchronous.
- Several state changes happening right after each other may be batched for performance reasons.

Was this helpful?

Functional state update

Last updated July 2022

Because state updates are asynchronous, there's a caveat that we have to be aware of.

For the sake of simplicity, let's say that we have the following component:

```
import {useState} from "react";

function App() {
    const [counter, setCounter] = useState(0);

    function handleButtonClick() {
        setCounter(counter + 1);
        setCounter(counter + 1);
    }

    return <button onClick={handleButtonClick}>Click me {counter}
</button>
}
```

This is just for demo purposes; weirdly, there are two `setCounter` calls after each other.

But assuming this code, what do you expect the counter's value will be after clicking once on the button?

The value will be 1 not 2, here's why:

```
//assuming: counter is 0
setCounter(counter + 1);
setCounter(counter + 1);
```

`setCounter()` call set the counter to `counter + 1`, but not immediately because it's asynchronous.

Then we have the following call `setCounter(counter + 1)`, but the value of `counter` is still 0 because the component did not re-render yet.

So the second call will also set the state to 1.

A reminder that this is caused because the state updates are batched.

Functional state updates

To solve this issue, React has the concept of **functional state updates** which is when you pass a **function** to the state update function, here's an example:

```
setCounter((previousCounter) => {
    return previousCounter + 1;
});
```

You can use a shorter version of course:

```
setCounter(previousCounter => previousCounter + 1);
```

Notice how we give it a function definition that takes the previous value of the state and returns the new state. In this example, the new state is the previous one + 1.

Here's how you can fix the above example to add to the state twice:

```
import {useState} from "react";

function App() {
    const [counter, setCounter] = useState(0);

    function handleButtonClick() {
        setCounter(prevCounter => prevCounter + 1);
        setCounter(prevCounter => prevCounter + 1);
    }
```

```
</button>
}
```

This will add 2 every time you click on the button.

You might be confused here about where the `prevCounter` is coming from. Remember that `prevCounter => prevCounter + 1` is a *function definition*. React will call this function and pass the previous value of the state as the **first argument**.
This means you can name that argument whatever you want. In this example, we used `prevCounter`.

Note that we've already created several counters. We did not have issues with the counter components because our app and component are so simple, and there's only one state update. However, it is a **recommendation** that:

Whenever the new state is computed using the previous state, then you should use functional state updates to guarantee consistency and prevent unexpected bugs.

So, if you're incrementing a counter, you should use a functional state update. On the other hand, if you're resetting a counter back to 0, you don't necessarily have to use a functional state update (as the new value is *not* computed using the previous state).

Recap

- `setState` function from `useState` can receive a function that allows you to perform functional state update.
- The function passed to `setState` receives the previous state as an argument you can use to compute the next state.
- Whenever the new state is computed using the previous state, then you should use functional state updates to guarantee consistency and prevent unexpected bugs.

Was this helpful?

Chapter recap

Last updated July 2022

Remember to use functional state updates whenever the new state is computed based on the previous state or props. This way, you avoid unexpected bugs in your application.

Recap

- State updates in React are asynchronous.
- Several state changes happening right after each other may be batched for performance reasons.
- `setState` function from `useState` can receive a function that allows you to perform functional state update.
- The function passed to `setState` receives the previous state as an argument which you can use to compute the next state.
- Whenever the new state is computed using the previous state, you should use functional state updates to guarantee consistency and prevent unexpected bugs.

Was this helpful?

Intro to useEffect

Last updated July 2023

Before we learn about the `useEffect` hook, we'd like to briefly introduce *React Strict Mode*.

React.StrictMode

React includes a `StrictMode` component that helps you find some unwanted bugs when React is running in *development mode*.
You can keep `StrictMode` in *production mode* as it has *no effect at all*.

Under the `React` object, you can access the `StrictMode` component. Since `React` is an object and `StrictMode` is a key of that object, you can access that component using the following syntax: `<React.StrictMode></React.StrictMode>` as long as you've imported `React`.

Here's an example:

```
import React from "react";
import {createRoot} from "react-dom/client";

function App() {
}

const root = document.querySelector("#root");

createRoot(root).render(<React.StrictMode><App />
</React.StrictMode>);
```

Alternatively, you can import `StrictMode` as a named import and then use it as follows:

```
function App() {
}

const root = document.querySelector("#root");

createRoot(root).render(<StrictMode><App />></StrictMode>);
```

StrictMode is often included by default in Vite, and React frameworks such as Next.js.

From now on, most challenges in this course will be wrapped with `React.StrictMode` to get you used to its behavior in *development mode*.

Strict mode will help you find a set of errors that are accidentally made when using the `useEffect` hook that we will learn about below. We'll explain how *strict mode* helps in a bit.

useEffect

We've seen so far one *react hook* which is `useState`. In the following few chapters, we'll deep dive into the `useEffect` hook and understand more about the concept of hooks.

The `useEffect` hook lets you perform an **effect** in your component.
Here are some examples of effects:

- Send a request to your analytics provider
- Initialize a DOM plugin outside of React (for example, drawing a map; we'll see that in the following Project)
- Change the page title (the one that shows up in the browser tab)
- Subscribe a user to a live chat service (with WebSockets)

They are called effects because they are instructions running outside your component or as a result of your component.

example, changing the page title is considered an external API that is not managed by React.

You should use effects sparingly. We'll discuss that more in the chapter recap.

There are two types of effects, those that don't require cleanup and those that do. In this chapter, we're only going to cover effects that do **not** require cleanup (as they are easier to get started with).

Page title

Let's start with the easiest and most common example of an effect that updates the page title.
In JavaScript, you can update the page title by changing the `document.title`, for example:

```
document.title = "Intro to useEffect | React Tutorial App";
```

If you look at the current page title in your browser, it will most likely be *Intro to useEffect | React Tutorial App*. You can update this title with `document.title`.

Now let's say we have a React component and we'd like to keep the title in sync with the current state; for example, we have a `counter` state, and we want to show that counter value in the title; here's how we can do it:

```
import {useState, useEffect} from "react";

function Counter() {
    const [counter, setCounter] = useState(0);

    useEffect(() => {
        document.title = `Counter is ${counter}`;
    });

    function handleButtonClick() {
```

```
    return <button onClick={handleButtonClick}>Click me {counter}
</button>
}
```

Notice how we imported `useEffect`; the syntax is similar to how you import `useState`.

Here's how you can import `useEffect` alone (without React and/or useState):

```
import {useEffect} from "react";
```

So `useEffect` is a named export from the `react` package.

Now notice how `useEffect` accepts a function definition as its first argument:

```
useEffect(() => {
    document.title = `Counter is ${counter}`;
});
```

Now this function:

```
() => {
    document.title = `Counter is ${counter}`;
}
```

will be called **after every** re-render of this `Counter` component.

So after every time the `Counter` component renders (or re-renders), the above function will be called, which will update the page's title.

This will allow you to synchronize the document's title with the `counter` value. Here's what it looks like:

Recap

- `<React.StrictMode />` helps you find unwanted bugs when React is running in *development mode*.
- React strict mode has **no effect** in *production mode*.
- The `useEffect` hook lets you perform an **effect** in your component.
- Effects are needed when you need to *synchronize* React with any external API.
- `useEffect` is a named export from the `react` package.
- `useEffect` accepts a function definition as its first argument, which will be run **after every** re-render of that component.

Was this helpful?

UseEffect deep dive

Last updated July 2022

Coming back to the previous example:

```
import {useState, useEffect} from "react";

function Counter() {
    const [counter, setCounter] = useState(0);

    useEffect(() => {
        document.title = `Counter is ${counter}`;
    });

    return <button onClick={() => setCounter(prevCounter =>
prevCounter + 1)}>Click me {counter}</button>
}
```

Let's break down what's happening to understand better how `useEffect` works.

When you call `useEffect`, you're telling React that your component needs to do something **after every** render.

React will store the function you passed, in this example, the one that updates the `document.title` and then will call it after completely being done with the DOM updates of this component.

Closures reminder

`useEffect` is placed **inside** the component on purpose so that you can easily use the existing variables there (most notably the state variables).

Notice that because `useEffect` is called inside the function `Counter`, you will be able to access all the variables in the parent function (`Counter`). We know that from closures. In this example, we could access the `counter` variable.

date value of that state. In this example `counter`.

Rules of hook

Remember the **rules of hooks** as they still apply!

Rule #1: Only call Hooks from React functions

You can only use `useEffect` inside a React function.

Rule #2: Only call Hooks at the Top Level

and never call hooks inside loops, conditions, or nested functions.

This is the most important and error-prone rule.

Just like we never wrapped `useState` with an `if` condition, you should never wrap `useEffect` with an `if` condition.
However, it's perfectly fine to have an `if` condition **inside** the useEffect function.

So this is **incorrect**:

```
// pseudo-code
function App() {
    if (someState ≥ 0) {
        useEffect(() => {
            // THIS IS INCORRECT 👎
        })
    }
}
```

Instead, the `if` condition should go **inside** the `useEffect` call:

```
// pseudo-code
function App() {
    useEffect(() => {
        if (someState ≥ 0) {
```

```
    });
}
```

Make sure to keep your hooks call at the top of the function to avoid unexpected errors.

Infinite loops

It's quite easy to create infinite loops while using the effect hook, but don't worry, as we have a feature baked into this course that will detect most infinite loops and stop them ;)

Recap

- Remember the rules of hooks as they still apply!
- Rule #1: Only call Hooks from React functions
- Rule #2: Only call Hooks at the Top Level and never call hooks inside loops, conditions, or nested functions.

Was this helpful?

Chapter recap

Last updated July 2022

Great work!

You might be wondering, why do we need a `useEffect` hook? Can't we just call `document.title = " ... "` anywhere in the component code before the `return` statement?

While you can, this is extremely discouraged for many reasons.
One of the main reasons is performance. Your components should be very fast to render, allowing your app to keep re-rendering very fast. Also, the code you have before the `return` statement runs *before* the JSX is produced from your component. On the other hand, code inside the `useEffect` hook runs only **after** your component has rendered to the DOM.

There's also another *very important* reason for that when it comes to effects that require clean-up. That's precisely what we'll cover in the following chapter. See you there!

Recap

- `<React.StrictMode />` helps you find unwanted bugs when React is running in *development mode*.
- React strict mode has **no effect** in *production mode*.
- The `useEffect` hook lets you perform an **effect** in your component.
- Effects are needed when you need to *synchronize* React with any external API.
- `useEffect` is a named export from the `react` package.
- `useEffect` accepts a function definition as its first argument, which will be run **after every** re-render of that component.
- Remember the rules of hooks as they still apply!
- Rule #1: Only call Hooks from React functions

Was this helpful?

Effect with clean up

Last updated May 2023

In the previous chapter, we saw how to run an **effect** that does not require cleanup. In this chapter, we'll explore effects that require cleanup.

What is cleanup?

Some JavaScript methods require cleanup; let's see why.
When you create a timer using `setTimeout`, you are delaying a piece of code to run in the future, for example:

```
setTimeout(() => {
    console.log("This will run in 1 second");
}, 1_000);
```

If this code is wrapped in a function and that function might be called more than once, you risk having a memory leak. This means you risk gradually increasing your application's memory because you keep scheduling a timer every time this function is called.

A more concrete example is when you run this code inside a React component, for example:

```
import {useEffect} from "react";

function App() {
    useEffect(() => {
        setTimeout(() => {
            console.log("This will run in 1 second");
        }, 1_000);
    });
```

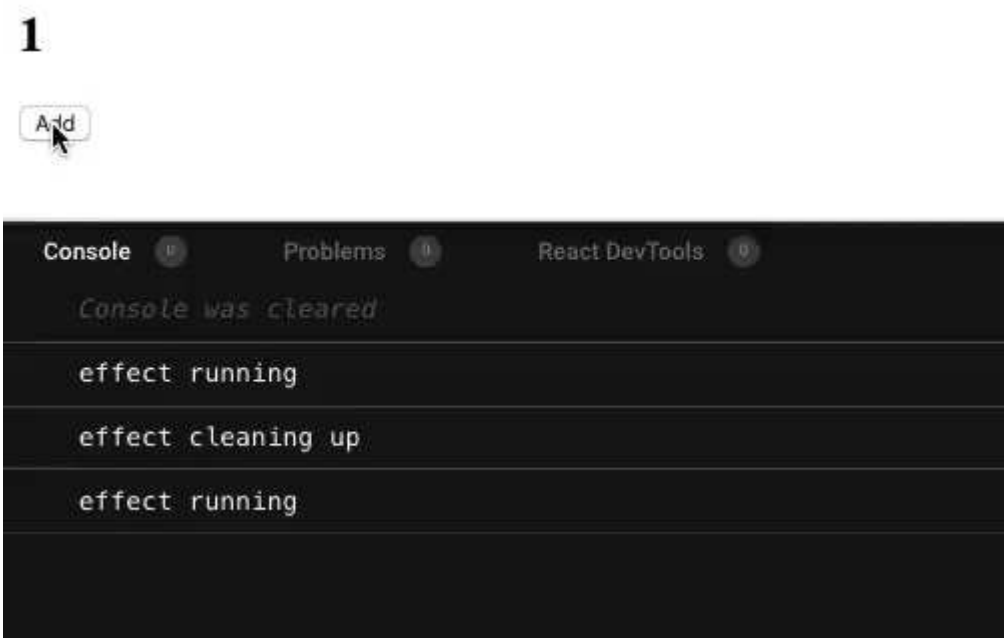
Every time the App component re-renders, it will schedule a new timer *without* cleaning up the previous timer.

This is an example of an effect that requires cleanup to avoid having a memory leak. This example only shows a `setTimeout` but imagine you're setting up a connection to a chat API using WebSockets without closing the previous one. This is another example of a potential memory leak which is also an example of an effect that requires cleanup.

How to clean up effects

In order to clean up an effect, you have to return a (clean up) function from inside the `useEffect` call. We'll start with the most basic example:

```
import {useEffect, useState} from "react";

function App() {
  const [counter, setCounter] = useState(0);

  useEffect(() => {
    console.log("effect running");
    return () => {
      console.log("effect cleaning up");
    }
  })

  return (
    <>
      <h2>{counter}</h2>
      <button onClick={() => setCounter(prevCounter =>
prevCounter + 1)}>Add</button>
    </>
  );
}
```



Notice how the **first** time the component renders, we only see the console.log: `"effect running"`.
Then **every time** the component re-renders because its state is changing (which is when we click on the button), then it starts by cleaning up the component (we see `"effect cleaning up"`) and then `"effect running"`.

In conclusion, `return () => { console.log("effect cleaning up"); }` will run every time the component is being removed (also called unmounted).
Note that it is being removed and then inserted again because it's re-rendering due to a state change.

In the next lesson, we'll see how to use that knowledge to clean up our timer.

Strict Mode

When running the code above in strict mode on a development environment, you will see the following in the console:

```
effect running
effect cleaning up
effect running
```

forget to clean up your effect, you should be able to see some unexpected behavior in your app (only when running locally in strict mode).

This behavior allows React's reconciliation algorithm to abort rendering halfway while ensuring that this doesn't break your UI. This paves the way to React 18's concurrent features (some of which are still in development).

When you run your code in a production environment (using React production), the effects will only run once, and you won't experience this behavior. So, it's only a local behavior that aims to help you find bugs. We'll have a visualization challenge next that helps you better understand what we mean by this.

Recap

- Some JavaScript methods require cleanup to avoid memory leaks (for example, `setTimeout`, `setInterval`).
- To clean up an effect, you have to return a function from inside the `useEffect` call.
- React will call that function every time the component is being re-rendered or removed from the DOM.
- If you forget to clean up your effect, you should be able to see some unexpected behavior in your app (only when running locally in strict mode).

Was this helpful?

Clean up timer

Last updated May 2023

Coming back to the previous example:

```
import {useEffect} from "react";

function App() {
    useEffect(() => {
        setTimeout(() => {
            console.log("This will run in 1 second");
        }, 1_000);
    });

    return <h1>App</h1>;
}
```

We need a way to clean up the timer. JavaScript provides the `clearTimeout(timerId)` function which takes the `timerId` as a parameter. The `timerId` can be stored by assigning the `setTimeout` call to a variable.

In plain JavaScript, here's how you create a timer and then cancel it immediately:

```
const timerId = setTimeout(() => {
    console.log("this is supposed to run in 1 second");
}, 1_000);

clearTimeout(timerId); // this cancels the timer
```

This ends up canceling the timer, and we won't get **anything** on the console.

Now we know that if we need to ask React to clean up a timer when the component is being re-rendered, then we have to return a function from `useEffect`. Let's do that:

```
import {useEffect} from "react";
```

```
    useEffect(() => {
        const timerId = setTimeout(() => {
            console.log("This will run in 1 second");
        }, 1_000);

        return () => {
           // when component gets unmounted, clear the timer
           clearTimeout(timerId);
        };
    });

    return <h1>App</h1>;
}
```

Every time the component gets unmounted (or re-rendered), React will call the function that we returned from `useEffect`.

> Be careful not to put the `return () => { ... }` inside the `setTimeout`. Your goal is not to return from the setTimeout but rather from the `useEffect`.

This sounds very inefficient...

You may wonder why React cleans up the effect after every re-render. It does sound inefficient. However, it helps you avoid memory leaks and minor bugs that are often hard to detect.

This cleanup also occurs when the component is removed from the DOM. We'll discuss that in the following chapter.

Optional: naming the cleanup function

It is also possible to give a name to the function that will clean up. This is entirely optional. The benefit of providing a name is reminding yourself that this function will only be called on cleanup. That name would also show up in the call stack when an error occurs – that's useful for debugging. You can also give a more descriptive name,

For example:

```
useEffect(() => {
    const timerId = setTimeout(() => {
        console.log("This will run in 1 second");
    }, 1_000);
    return function cleanupTimer() {
        clearTimeout(timerId);
    };
});
```

This is entirely optional, and it's up to you to decide whether you want to name the functions or not.

Cleanup vs. no cleanup

The difference between an effect that will clean up and an effect that will not clean up is just whether you return a function to clean up or not.

So effects that do **not** return a function are the effects that React will not clean up. Whereas effects that return a function are the effects that React will clean up on re-render.

How does it work?

React will store the function definition you return from `useEffect` when it renders your component. In our example, that's:

```
() => {
    clearTimeout(timerId);
};
```

Remember from the closures chapter that the value of `timerId` is accessible inside this function because it's a closure inside its parent function.

the DOM).

Cleanup runs after every re-render

The reason why the cleanup function is called after **every** re-render is to avoid bugs. It's a design decision that aims to reduce the number of bugs and memory leaks. However, in some scenarios, you may need to run the `effect` and the cleanup on specific cases. This will be discussed in the next chapter.

Recap

- `const timerId = setTimeout(() => {}, 1_000)` will store the `timerId` which allows you to clean it up later on.
- `clearTimeout(timerId)` cancels the timer initiated by `setTimeout`.
- You can give an optional name to the cleanup function.
- Effects that do NOT return a function are the effects that React will not clean up.
- Effects that return a function are the effects that React will clean up on re-render.
- The cleanup runs after every re-render. You will be able to prevent that from running on every re-render. We'll see that in the next chapter.

Was this helpful?

Event listeners

Last updated October 2023

The effect hook is an advanced topic in React; thus, we're breaking it into several chapters. The examples we're seeing so far are not the most exciting. We will get to the more exciting examples very soon. For example, updating the state inside the effect hook (but don't do that just yet because it'll cause an infinite loop!).

Event listener outside React

We've seen how you can add an event listener to an element with the `onClick` (or similar) JSX attribute. But what if you need to add an event listener to the `window` object?
That's outside the React world and is thus considered an effect. That's because when your component renders, it has a side effect on an outside element, the `window` object. That side effect is the fact that there's a new event listener.

This could be useful if you want to know when the user scrolls inside your page for some reason.
Here's how we'd implement that:

```
import {useEffect} from "react";

function App() {
  useEffect(() => {
    window.addEventListener("scroll", () => {
        console.log("page scrolled");
    });
  });

  return (<>
    <h2>List of products</h2>
    <p>Lorem ipsum ... </p>
```

This works but has a big problem. It's making it possible to have a memory leak because every time the App component is re-rendered, we end up adding a *new event listener*.

This is why whenever you call `addEventListener` inside `useEffect`, you need to clean up afterward by calling `removeEventListener`.

Cleaning up the event

In order to clean up the event, we need to give a function name to the event listener. In plain JavaScript (without React), here's how you add and then remove an event:

```
// we need to give a name to the event so that we can remove it
later on
function handleWindowScroll(event) {
    console.log("page scrolled");
}

// call the handleWindowScroll function when the page is scrolled
window.addEventListener("scroll", handleWindowScroll);

// remove the event
window.removeEventListener("scroll", handleWindowScroll);
```

So now going back to useEffect, we need to return a function from useEffect to cleanup this event listener:

```
import React, {useEffect} from "react";

function App() {

  function handleWindowScroll(event) {
    console.log("page scrolled");
  }
```

```
    window.addEventListener("scroll", handleWindowScroll);

    // cleanup event listener
    return () => {
        window.removeEventListener("scroll", handleWindowScroll);
    }
  });

  return (<>
    <h2>List of products</h2>
    <p>Lorem ipsum ... </p>
  </>);
}
```

Now when the App component is unmounted (removed from the DOM) and every time it re-renders, React will clean up the event listener avoiding memory leaks.

Note that if you want to add an event to an element described in your JSX, you still have to use the `onClick`, `onChange`, etc., syntax. This is only meant for adding events to the `window`, `document` objects, or objects outside of React.

Recap

- When adding an event listener outside React, remember to clean it up
- You can clean up an event listener with `removeEventListener`, which requires two arguments: the event's type and the name of the function handling that event.

Was this helpful?

Chapter recap

Last updated July 2022

Great work!

In the following chapter, we'll look at how you can decide **when** to run the effects.

Your feedback about this course

Chapter recap

- Some JavaScript methods require cleanup to avoid memory leaks (for example, `setTimeout`, `setInterval`).
- To clean up an effect, you have to return a function from inside the `useEffect` call.

- If you forget to clean up your effect, you should be able to see some unexpected behavior in your app (only when running locally in strict mode).
- You can give an optional name to the cleanup function.
- Effects that do NOT return a function are the effects that React will not clean up.
- Effects that return a function are the effects that React will clean up on re-render.
- The cleanup runs after every re-render. You will be able to prevent that from running on every re-render. We'll see that in the next chapter.
- When adding an event listener outside React, remember to clean it up

Was this helpful?

Dependencies array

Last updated October 2023

All the `useEffect` calls we've seen so far run after the first time the component has rendered and after every re-render. But sometimes you don't want that `useEffect` to re-render every time (sometimes this will create infinite loops!). This is why the `useEffect` function takes a **second argument**.

Here's the function signature of `useEffect`:

```
useEffect(effectCallback, dependencies)
```

The `effectCallback` is the function we pass as the first argument, and the `dependencies` argument will be explained in this chapter. It defaults to `null`. So in the previous chapters, we never passed that parameter; thus it has defaulted to `null`.

useEffect dependencies

The `dependencies` in `useEffect` is an array that will dictate **when** to re-run the effect. Here's how it works:

```
import {useState, useEffect} from "react";

function App() {
    const [counter, setCounter] = useState(0);

    useEffect(() => {
        console.log("effect is running");
    }, [counter]);

    return <button onClick={() => setCounter(prev => prev +
1)}>Click me</button>;
}
```

dependencies.

This tells React that it should **only call the useEffect again** when the value of `counter` changes. This means it will call after the first time the component has rendered and every time the `counter` value changes.

So in this example, `[counter]` starts off as `[0]` (because the state `counter` starts at `0`).
React will store this `[0]` and then when the user clicks on the button.
The new value of `[counter]` becomes `[1]`.
So React will then compare this new value to the old one (one by one). And because 1 in `[1]` is not the same as the `0` in `[0]` (from the previous run), then React will **re-run** the useEffect.

The dependency is an array because you could have several values there (from state, props, or any variable defined inside the component). We'll get to these examples later in the course.

This allows you to optimize your code and make it run only when necessary.

Run only once

It's also possible to pass an empty array as a dependency: `[]`, and in fact, this is quite common.
Because it's an empty array, the comparison between the previous render and the next render **will always be the same** (since React is comparing the same instance of `[]`, which doesn't change). This means that passing an `[]` will **only run the effect once** after the first render.

So if you ever want to run the effect just once, after the component has been mounted, you should pass an empty array. This is often useful when working with libraries, or sending analytics calls in a component.
Let's take the example of drawing a map using an external library, which is the following project!

In JavaScript, you need to initiate a map library just once by calling the below pseudo-code:

```
    container: "#map",
    style: "mapbox://styles/mapbox/streets-v11",
    center: [10, 20],
    zoom: 13
});
```

But you don't need to re-run it every time the state of that component changes! You only need to run it once. This is why in React, you need to run it inside `useEffect` and with an empty array as a dependency. Here's the pseudo-code for that:

```
useEffect(() => {
    const map = new mapboxgl.Map({
        container: "#map",
        style: "mapbox://styles/mapbox/streets-v11",
        center: [10, 20],
        zoom: 13
    });

    // no need to clean up as the map is destroyed when the DOM
element is removed
}, []);
```

Recap

- The function signature of useEffect is: `useEffect(effectCallback, dependencies)`
- By default, the dependencies argument is `null`, which means it will re-run on every update.
- The dependencies parameter accepts an array of values that React will compare from the previous render to the next render. Only when they change then React will call the effect again.
- If you pass an empty array `[]` as a dependency, then the effect will only be called once.

Component lifecycle

Last updated February 2023

With `useEffect` and its dependency, you can call methods at different life cycles of the component. This means you can call methods at different **stages** of the component. For example:

- after the first time when it mounts (gets added to the DOM)
- after every time it re-renders
- after the component is unmounted (removed from the DOM)
- after a re-render, when a certain state variable changes

You already know how to create all of them, but since they're important, we'd like to highlight them.
Note: if you already know how to work with React in **classes** (before version 16.8), then we'll show you the `name` of the old life cycle hook so you can mentally understand how to migrate to hooks. If you never worked with classes, please forget about it and don't worry about it! We'll cover it in the Optional classes chapter.

Once after component is mounted

To run an effect **once** after the component has mounted (was added to the DOM), you need to call `useEffect` with an **empty array** as the list of dependencies.

```
import {useEffect} from "react";

function App() {
    useEffect(() => {
        console.log("Once after component mounted")
    }, []);
}
```

If you know about React classes, this is the equivalent to `componentDidMount`.

> Don't forget that your components are mounted, unmounted, and then re-mounted again in Strict mode when running React in development mode.

Once after component is mounted + once before unmounting

This is going to be similar to above but with a **cleanup**. Thus we'll return a function from `useEffect`:

```
import React, {useEffect} from "react";

function App() {
    useEffect(() => {
        console.log("Once after component mounted")

        return () => {
            console.log("Once before component unmounted");
        }
    }, []);
}
```

If you know about React classes, the function returned is the equivalent to `componentWillUnmount`. So this example shows `componentDidMount` and `componentWillUnmount`.
This is also useful for some libraries/plugins that require cleaning up (such as a chat library) as well as functions that have side effects such as `setTimeout` and `setInterval`.

On every re-render

This allows you to run the effect after **every re-render** as well as after the first render (which is when the component mounts):

```
import {useEffect} from "react";
```

```
    useEffect(() => {
        console.log("After it mounted the first time and after
every re-render");
    });
}
```

If you know about React classes, this is similar to `componentDidUpdate` **and** `componentDidMount` together.

This is useful to update the `document.title` using a certain state variable inside that component.

On certain re-renders

Sometimes you want to run the effect only when a certain state variable changes or props (or both together). This can be achieved by passing that value to the list of dependencies:

```
import {useEffect, useState} from "react";

function App(props) {
    const [first, setFirst] = useState(0);
    const [second, setSecond] = useState(0);

    useEffect(() => {
        console.log("Only when first and/or props.user change.")
    }, [first, props.user]);
}
```

This will instruct React to only call this effect again when at least one of the two values changes: `first` or `props.user`.
Note it will also run the first time the component has rendered.

If you know React classes, this is similar to `componentDidUpdate` but with an additional `if` condition that checks the previous value of the `first` state and the `user` prop.

You can have several effects in the same component, and each effect could have a different dependency. Just make sure to keep in mind the rules of hooks and keep all your hooks at the top of the function and never wrap them with if conditions.

Recap

- You can run an effect:

1. Once after component is mounted
2. Once after component is mounted + once before unmounting
3. On every re-render
4. On certain re-renders

Was this helpful?

Chapter recap

Last updated March 2023

We can finally explain why `hooks` are called that way.
From the official documentation: Hooks are functions that let you "hook into" React state and lifecycle features from function components.

For example, `useEffect` "hooks" into the component's lifecycle (when it mounts, unmounts, etc.).
The `useState` hook hooks into React state features and allows you to set state.

useEffect is an escape hatch

As of React 18, the `useEffect` hook is now considered an escape hatch. You shouldn't be using it too often. Also, its most common use case is synchronizing with an external API.

So, you shouldn't find yourself using `useEffect` that often.

At the moment, the course uses `fetch` inside `useEffect`. React is currently working on a new `use` hook. Once this hook makes it into the stable version, we'll update the course to use this new hook with `fetch`. We've added a video screencast in the meantime in chapter 57 showcasing this hook.

Recap

- The function signature of useEffect is: `useEffect(effectCallback, dependencies)`
- By default, the dependencies argument is `null`, which means it will re-run on every update.

compare from the previous render to the next render only when they change then React will call the effect again.

- If you pass an empty array [ ] as a dependency, then the effect will only be called once.
- You can run an effect:

1. Once after component is mounted
2. Once after component is mounted + once before unmounting
3. On every re-render
4. On certain re-renders

Was this helpful?

Integrating a map

Last updated July 2022

This project will be a basic introduction to integrating a map in your web app using React.

Once we learn about state inside `useEffect`, we'll have another advanced map project ;)

Mapbox

We'll be using Mapbox because it allows us to add a map to our website for free (and without having to add your credit card number).
Details on how to set up an account will be in the challenge itself. The reason why you need an account is that you need an **access key**, which is a string that lets Mapbox

On free accounts, you have around 50,000 map loads per month. Check out their pricing page if you're interested to learn more.

Creating the map

To create the **map**, we need to call `mapboxgl.Map`. The exact details will be given in the next challenge. The code in this lesson is just pseudo-code.

```
new mapboxgl.Map({
    container: "map",
    center: [10, 5],
});
```

This will create a map and center it at the point where the longitude is 10, and the latitude is 5.
Note this `container: "map"`.
This will end up calling `document.querySelector("#map")`, which will be the DOM element where the map will be added.

Note that `"map"` as the `container` is not the best approach. We can do better with `refs`, but we haven't learned about them yet, so stay tuned!

Cleanup?

This example does not require any cleanup because as soon as the `<div id="map">` `</div>` is removed from the DOM (when your component is unmounted), then Mapbox will do the necessary cleanups. However if you add events with the `map.on("event")` API, then remember to clean them up with `map.off("event")` by returning a function inside `useEffect`.

We'll see that later in this course!

Some packages allow you to use Mapbox with React. They're often described as "React binding for Mapbox". For example: react-mapbox-gl.

There are some downsides when using such packages:

- They add weight to your application (more JavaScript KBs, which slow down your website)
- They make it harder for you to solve the issues you run into. That's because they're wrapping a library, and you need to consult that original library for those issues.
- Most notably, for now, we'd like you to learn how things work rather than how just to use packages.

Remember that it doesn't mean you should never use these types of packages. It's up to you and depends on what you're trying to build. Sometimes the package makes it a breeze to use external libraries; in that case, go for it! Make sure you understand the underlying library's workings before using a high-level abstraction.

Was this helpful?

Project recap

Last updated July 2022

Congratulations on completing the first map project!

We're going to learn how to add markers and update their positions in the next project, but first, we need to learn a few more concepts about `useEffect` when combined with `useState`, so keep going! 💪

Was this helpful?

Change state in effect

Last updated February 2023

In general, you are encouraged *not* to update your state from within `useEffect`. However, there may be some scenarios where this is necessary. In that case, we'd like to show you the most common error that you may encounter, which you are encouraged to make so that you understand it:

```
import {useState, useEffect} from "react";

// THIS IS INCORRECT
function App() {
    const [count, setCount] = useState(0);

    useEffect(() => {
        // this will end up creating an infinite loop
        setCount(previousCount => previousCount + 1);
    });
}
```

The code above is wrong because it ends up creating an infinite loop. Don't worry; if you make such a mistake on React Tutorial, we'll detect it and pause it ;)

Why does this happen?
The reason why you get an infinite loop is that:

1. when you call `setCount()` with a new value, it will end up re-rendering the component.
2. which means it will call the App component again.
3. which means the `effect` will run again. Which will call `setCount()` again and so on.

The fix for this depends on what the component is supposed to do. Still, the fix involves passing a dependency to the effect, which limits the effect from running only on certain conditions or making it run only once.

state variable. For example, let's say you've got a state storing the list of chapters. If you only want to render *completed* chapters (chapters that have `isCompleted` `===` `true`), then you don't need to create a new state variable for that. You can just create a variable that is computed from the original state variable. Here's how:

```
import {useState} from "react";

const exampleData = [{
  id: 1,
  title: "Strings",
  isCompleted: true
}, {
  id: 2,
  title: "Numbers",
  isCompleted: false
}];

function App() {
  const [chapters, setChapters] = useState(exampleData);

  const completedChapters = chapters.filter(chapter ⇒
chapter.isCompleted);

  return <ul>
    {completedChapters.map(chapter ⇒ <li key={chapter.id}>
{chapter.title}</li>)}
  </ul>;
}
```

Recap

- In general, it's best to avoid setting state inside `useEffect`.
- calling `setState` inside a `useEffect` will most likely lead to an infinite loop. You will have to pass a dependency to `useEffect`.

Was this helpful?

Building a clock

Last updated May 2023

Let's say we want to build a `<Clock />` component that shows the current time. We'd start by creating a state called `date`, and we give it as an initial value a new instance of `Date`. So that would be `new Date()`.
This will give us a `date` object which we can then call: `.toLocaleTimeString()` on and it will return to us the time, for example: `09:08:43` (*hh:mm:ss* format).

If you want to try the code in JavaScript before you get to React, here's what it looks like:

```
const date = new Date();
console.log(date.toLocaleTimeString()); // 09:08:43 (depending on
the time of the day)
```

Let's create our React component that will do the same:

```
import {useState, useEffect} from "react";

function Clock() {
    const [date, setDate] = useState(new Date());

    return <h2>{date.toLocaleTimeString()}</h2>
}
```

Now, this component will show the date at the time it was rendered and will never update.
We need to update the `date` state every 1 second to a new instance of `Date`.

So in JavaScript:

```
let date = new Date();
console.log(date.toLocaleTimeString());
```

```
    console.log(date.toLocaleTimeString()); // shows the time
every second
}, 1_000);
```

Note: `setInterval` allows you to call a function every X milliseconds. In this example, we gave it 1000 milliseconds so that it will call the function body every 1 second. However, it is not 100% accurate; if your browser was busy with other tasks, this could be deferred a little bit more than 1000 milliseconds. So our clock will not be perfect.

Now with React, we'll make the component call `useEffect` to initialize the interval:

```
useEffect(() => {
    setInterval(() => {
        // set a new instance of date every second
        setDate(new Date());
    }, 1_000);
    // TODO: still need to cleanup the side effect
}, []);
```

Notice how we set the `useEffect` to run just once by providing it with an `[]` (empty array) as the second parameter.
You can think of it as the following:

1. React renders the `<h2></h2>` with the time.
2. After rendering the first time, it will run the function inside useEffect only **once**
3. That function calls `setInterval` which will schedule the `setDate(new Date())` to be called once **every second**.
4. Every time `setDate()` is called, the component will re-render, showing the updated time on the screen.

Cleaning up the effect

The code above creates a memory leak because we're not clearing the interval we create with `setInterval`. Here's how you'd do that in JavaScript:

```
console.log(date.toLocaleTimeString());
// save intervalId in a variable so we can clear it later on
const intervalId = setInterval(() => {
    date = new Date();
    console.log(date.toLocaleTimeString());
}, 1_000);

// when you want to clear the interval:
clearInterval(intervalId);
```

Recap

- `const date = new Date();` creates a new instance of **Date**.
- `date.toLocaleTimeString()` shows the current time in `hh:mm:ss` format (potentially slightly different depending on your language/locale configuration).
- `setInterval` schedules a function to be called every X milliseconds.
- `clearInterval(intervalId)` allows you to clear an interval created with `setInterval`.

Was this helpful?

Flag for running effect

Last updated June 2024

Another common use case is having the ability to run an effect only when a certain variable is on or off. For that, we create a *boolean* state variable, and we use it as an if condition inside the effect.

Think of it as a **start** and **stop** button, which starts and stops the effect.

Let's take an example of a `Stopwatch`:

```
function Stopwatch() {
    const [counter, setCounter] = useState(0);

    useEffect(() => {
        let timerId = setTimeout(() => {
            setCounter(prevCounter => prevCounter + 1);
        }, 1_000);
        return () => {
            clearTimeout(timerId)
        }
    });

    function handleButtonClick() {
        // TODO
    }

    return <>
        <h2>{counter}</h2>
        <button onClick={handleButtonClick}>Start /
Pause</button>
    </>
}
```

Here's how it works:

1. This Stopwatch component will start from 0 (initial value of `useState()`).

3. The `useEffect()` is called, which schedules a timer for 1 second in the future.
4. After 1 second has elapsed, we call `setCounter(prevCounter => prevCounter + 1)` which increments the value of the counter by 1.
5. This ends up re-rendering the component, showing 1 and the `useEffect` runs again.
6. And so on.

How can we make the `Start / Pause` button work?

Start / Pause button

For that, we'll need a `state` variable that is a boolean:

```
const [running, setRunning] = useState(false);
```

We will use the value of `running` to decide whether the `useEffect` will run or not.

But note that from the [rules of hooks](), we are **NOT** allowed to wrap the `useEffect` with an `if` condition.

This is why we should wrap the code **inside** the `useEffect`:

```
useEffect(() => {
    // the effect still runs every time
    // but the code inside of it only runs when 'running' is true
    if (running) {
        let timerId = setTimeout(() => {
            setCounter(prevCounter => prevCounter + 1);
        }, 1_000);
        return () => {
            clearTimeout(timerId)
        }
    }
});
```

We use the logical not operator (explained in this lesson under inverting a boolean).

So the function `handleButtonClick` becomes:

```
function handleButtonClick() {
    setRunning(prevValue => !prevValue);
}
```

Recap

- You can create a boolean state variable to decide when to run an effect.
- You should NOT wrap the useEffect with an `if` condition but rather have the `if` condition inside the `useEffect`. This is because of the rules of hooks.

Was this helpful?

Chapter recap

Last updated May 2023

Let's take a step back because we're finally at a moment where the value of React should be obvious.

Look at this `<Stopwatch />` component in React:

```
import {useState, useEffect} from "react";

function Stopwatch() {
    const [counter, setCounter] = useState(0);
    const [running, setRunning] = useState(false);

    useEffect(() => {
        if (running) {
            let timerId = setTimeout(() => {
                setCounter(prevCounter => prevCounter + 1);
            }, 1_000);
            return () => {
                clearTimeout(timerId)
            }
        }
    });

    function handleStopClick() {
      setCounter(0);
      setRunning(false);
    }

    return <>
        <h2>{counter}</h2>
        <button onClick={() => setRunning(prevRunning =>
!prevRunning)}>Start / Pause</button>
        <button onClick={handleStopClick}>Stop</button>
    </>
}
```

```
let counter = 0;
let running = false;

function tick() {
  counter += 1;
  document.querySelector("#counter").textContent = counter;
}

let intervalId = setInterval(tick, 1_000);

document.querySelector("#start-pause").addEventListener("click",
() => {
  running = !running;
  if (running) {
    intervalId = setInterval(tick, 1_000);
  } else {
    clearInterval(intervalId);
  }
});

document.querySelector("#stop").addEventListener("click", () => {
  clearInterval(intervalId);
  counter = 0;
  running = false;
});
```

Even though they both work, the React one is slightly shorter and easier to read. If it's still not easy to read, that's because you recently learned about these concepts. Still, if you revisit this a week from now, you will notice that it has become easier.

Note that lines of code don't really matter, the goal is not to solve a challenge in the least amount of lines of code, but the goal is to have reusable, easy-to-maintain code.

And React here has achieved this job very well. As a side note, React is not the only tool to achieve that job.

Chapter Recap

- calling `setState` inside a `useEffect` will most likely lead to an infinite loop. You will have to pass a dependency to `useEffect`.
- `const date = new Date();` creates a new instance of **Date**.
- `date.toLocaleTimeString()` shows the current time in `hh:mm:ss` format (potentially slightly different depending on your language/locale configuration).
- `setInterval` schedules a function to be called every X milliseconds.
- `clearInterval(intervalId)` allows you to clear an interval created with `setInterval`.
- You can create a boolean state variable to decide when to run an effect.
- You should NOT wrap the useEffect with an `if` condition but rather have the `if` condition inside the `useEffect`. This is because of the rules of hooks.

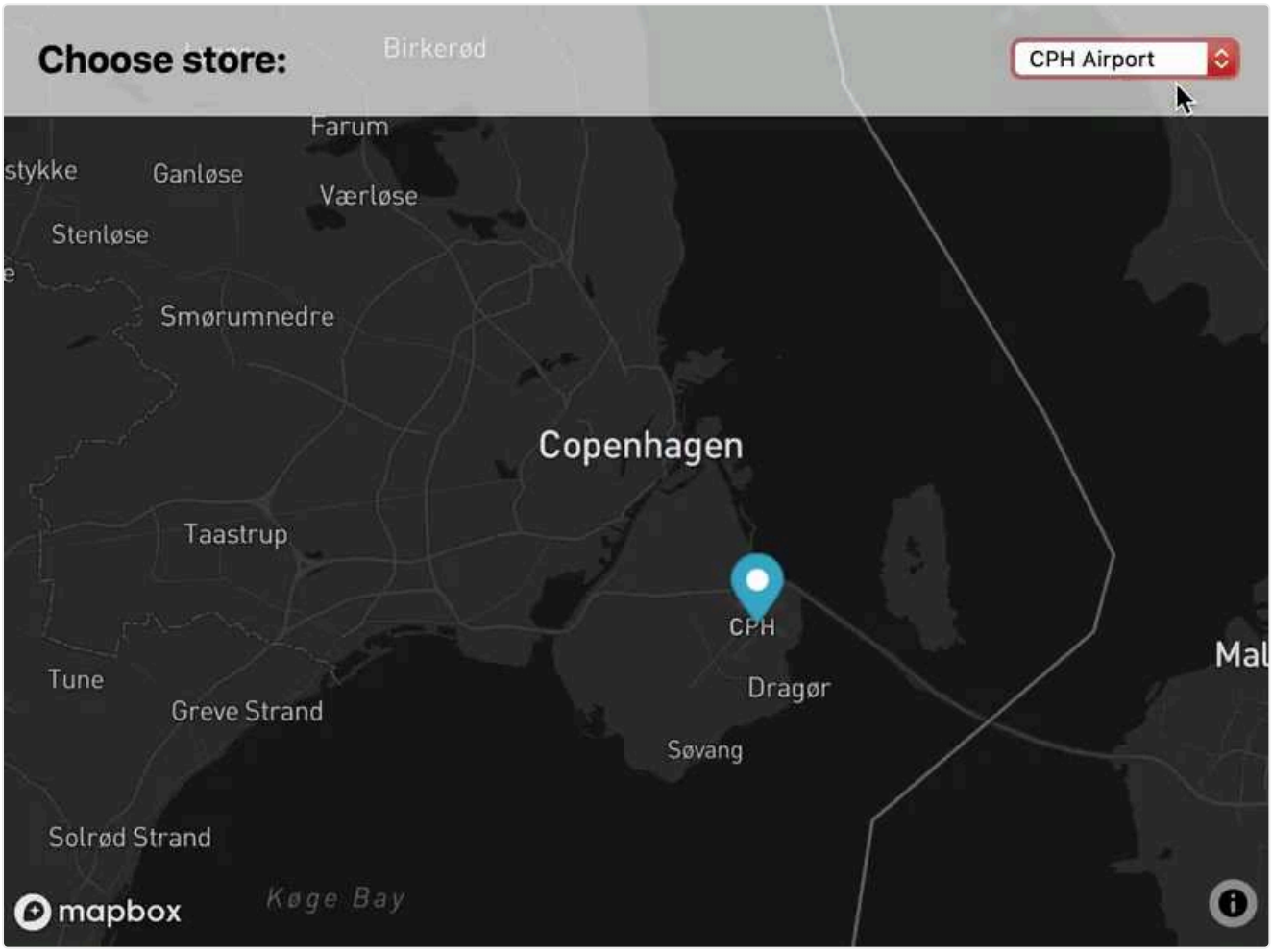
Was this helpful?

Store locator

Last updated July 2022

We'll enhance the previous Map project by adding a store locator dropdown in this project.

The dropdown contains 3 example locations in Denmark: Central station, Norrebro street, and CPH Airport. When the user selects a store from the dropdown, we will move the pin to the new coordinates.



Adding a marker

To add a marker, you need to have access to the `map` variable returned by `new mapboxgl.Map({})`.

We already have access to the map variable in that callback.

```
useEffect(() => {
    const map = new mapboxgl.Map({
        container: 'map',
        style: 'mapbox://styles/mapbox/dark-v10',
        center:  [ 12.567898, 55.67583 ],
        zoom: 9
    });
    // create marker
    const marker = new mapboxgl.Marker()
    .setLngLat([12.567898, 55.67583])
    .addTo(map); // add it to the existing map
}, []);
```

It also makes sense because this `useEffect` is only supposed to run **once**, and we only want to create the marker **once**.

The example above adds a marker to the longitude and latitude: 12.567898, 55.67583, which happens to be the same as the map center. Notice how we need to call `.addTo(map)`, which requires us to have access to the `map` variable. So if you were to call the marker instantiation outside the `useEffect`, then the `map` variable would be undefined.

Changing a marker's position

The map is created **once**. The marker is also created **once**, but every time the user changes the value from the dropdown, we need to change the position of the marker (rather than create a new one).

Here's how we can do that in JavaScript:

```
// assuming we have access to the 'marker' variable
 marker.setLngLat([10, 20]);
```

But this means that throughout our component, we need to have access to the `marker` variable so that we can call `setLngLat` when the user selects a value from the dropdown.
With the code above, `marker` is only available inside the `useEffect`.

How do we make that possible?

Setting the marker as state

There are several solutions for that; one of them is creating a variable `let marker` at the top of the component and then re-assigning it once we create the marker. This works because of how closures work, but here's the approach we recommend:

Think of the marker as being a state in this component.
Because it starts with being undefined, and then we will *set* the marker to the result we get back from `new mapboxgl.Marker()`, which will then be available inside our component.

So we start by defining the state for `marker`:

```
const [marker, setMarker] = useState(); // default value is
undefined
```

And then when we create the marker, we call `setMarker`:

```
const marker = new mapboxgl.Marker()
.setLngLat([12.567898, 55.67583])
.addTo(map);

setMarker(marker); // make it available throughout the component
```

This will allow you to use the `marker` variable throughout the whole component.

Was this helpful?

Project recap

Last updated July 2022

Congrats on completing this project!

Using your existing React knowledge to integrate a third-party library (such as Mapbox) is a little challenging. There will always be some learning curve when implementing third-party libraries. So it's perfectly normal if you feel that the last project was challenging!

Was this helpful?

Effect, State & Events

Last updated July 2022

There is a common confusion when it comes to using *effects*, *state*, and *events* all at the same time, so we have a chapter to cover that and explain why it happens.

Let's say we have the App component below:

```
import {useState} from "react";

function App() {
    const [count, setCount] = useState(0);

    function handleButtonClick() {
        setCount(prevCount => prevCount + 1);
        console.log(count);
    }

    return <>
      <h1>{count}</h1>
      <button onClick={handleButtonClick}>Add 1</button>
    </>;
}
```

This component shows the current `count`, starting at `0`.
When you click the `Add 1` button, it will add 1 to the `count` and then `console.log(count)`.
However, that `console.log(count)` will show us the **old count**. That's because React's state updates are asynchronous. This means that several state updates may be merged together and that you cannot rely on the value of the state to be up to date immediately after you call `setState`.

It's also because of the design of Hooks; you will only get the new value of `count` when the function **App** has been called again, which will call `const [count, setCount] = useState(0);` and give you the new up to date value of `count`.

The fix

If you need to log or use the updated value, that is of course, possible. All you need to do is use an *effect* with the dependency being that state variable.

That will make the effect run whenever the state variable changes. So here's how we fix our code above:

```
import {useState, useEffect} from "react";

function App() {
    const [count, setCount] = useState(0);

    useEffect(() => {
        console.log(count)
    }, [count]);

    function handleButtonClick() {
        setCount(prevCount => prevCount + 1)
    }

    return <>
      <h1>{count}</h1>
      <button onClick={handleButtonClick}>Click me</button>
    </>;
}
```

Notice how the effect:

```
useEffect(() => {

}, [count]);
```

the state count has a new value.

Final note

This code change will end up logging `0` when the component renders the first time. This behavior would not have happened in the previous code.

If that's something you don't want then you can wrap the console.log with an **if** condition:

```
useEffect(() => {
    if (count > 0) {
        console.log(count);
    }
}, [count]);
```

Recap

- React's state updates are asynchronous.
- If you need the updated value of a state that gets changed in an *event*, then you need to move the logic into an *effect*.

Was this helpful?

Chapter recap

Last updated July 2022

Great! In the following chapters, we'll learn about the performance of useEffect as well as how to use localStorage to store the products that the admin creates.

Chapter Recap

- React's state updates are asynchronous.
- If you need the updated value of a state that gets changed in an *event*, then you need to move the logic into an *effect*.

Was this helpful?

Effect Performance

Last updated July 2022

We've been using `useEffect` for a while now but haven't really covered its impact on performance.

The effect you schedule with useEffect will run **after** the component has rendered to the DOM, which is a choice consciously made by the React team for performance reasons. In this lesson, we'll see why.

localStorage example

`localStorage` (or `window.localStorage`) is a browser API that lets us store data made up of keys and values.
The problem with `localStorage` is that it is a synchronous API which means:

```
console.log("A");
localStorage.setItem("key", "value");
console.log("B");
```

In the code above, we will see the `console.log("A")`, then if `localStorage.setItem()` takes 200 milliseconds to complete, then the `console.log("B")` will have to wait for `localStorage.setItem()` to complete.

This is bad because it can immediately slow down your application.

If you have to store a few items, it is okay to use localStorage, but in general, you should not be using it very often due to its slow performance.

localStorage inside a component

should do it inside a `useEffect` for two reasons:

1. It is a side effect (it changes something outside the React component)
2. For performance reasons (illustrated below).

To illustrate why it's bad for performance, we set up a big array of objects in `data.js` that we import and then save into `localStorage` inside a React component. Here's the code:

```
import {useState} from "react";
import data from "./data.js";

function App() {
    const [random, setRandom] = useState(Math.random());

    // THIS IS BAD FOR PERFORMANCE
    localStorage.setItem("data", JSON.stringify(data));

    return <>
        <h1>{random}</h1>
        <button>Re-render</button>
    </>
}
```

The problem with the code above is that if the `localStorage.setItem()` takes 200ms, this will delay showing the DOM updates on the user's screen, which is terrible for performance.

To illustrate that, we took the example above and exaggerated it with 200 `localStorage.setItem()` and `localStorage.getItem()` calls.

And then we made an optimized version of it that has these 200 `localStorage` calls inside a `useEffect`, like this:

```
import {useState, useEffect} from "react";
import data from "./data.js";
```

```
    // this is BETTER for performance
    useEffect(() => {
        localStorage.setItem("data", JSON.stringify(data));
    });

    return <>
        <h1>{random}</h1>
        <button>Re-render</button>
    </>
}
```

By wrapping the `localStorage` calls with `useEffect`, we can let the component render to the screen **first** and, then execute the `localStorage` calls after it has rendered.

Here's what the exaggerated version looks like (we exaggerate it because you often start seeing performance issues when you have a large number of components, so by increasing the number of `localStorage` calls, we can simulate that).

200 localStorage calls directly inside the Component:

200 localStorage calls inside useEffect:

Notice how in the first video, the button remains **active** (highlighted in red color) because it is stuck while processing all the JavaScript code.

Note that `useEffect` is not a magic pill; if the code inside the `useEffect` is really slow, then the performance may still suffer.

Recap

- The effect you schedule with useEffect will run **after** the component has rendered to the DOM.

Was this helpful?

Chapter recap

Last updated July 2022

Now that we know that localStorage should be only used for small read/writes and that it should be wrapped with useEffect, in the next 2 chapters, we will learn how to store our state data to localStorage and how to restore it from localStorage!

Chapter Recap

- The effect you schedule with useEffect will run **after** the component has rendered to the DOM.

Was this helpful?

Storing into localStorage

Last updated July 2022

LocalStorage is a Web API that lets us store key-value pairs in the browser. It is limited to approximately 5MB. In the previous chapter, we saw that the localStorage API is synchronous, which could slow down our app. We should use it sparingly and inside `useEffect`.

The localStorage API can only save **strings**. So the key and the value will be strings. This is why we're going to see how to store strings, numbers, and booleans in this lesson. And in the next lesson, we'll see how to store arrays & objects.

In this chapter, we'll learn how to save our state to the localStorage. In the following chapter, we'll learn how to restore from localStorage (as well as how to do that efficiently).

Saving strings

Saving strings is straightforward because the localStorage API saves the key and value in string format. Here's an example:

```
let name = "Sam"

// Key is "name", value is the variable name (which has the value
"Sam")
localStorage.setItem("name", name);
```

You can then read it with `localStorage.getItem("name")`, and it will return the string: "Sam".

Saving numbers

Numbers will be saved as strings, meaning you need to parse them into a number when you reread them.

```
localStorage.setItem("age", age);
```

Later on, when you want to read it, you will get a **string** which means to get the number value, you have to convert it to a number:

```
let age = localStorage.getItem("age");
console.log(age); // "20" (string)
age = Number.parseInt(age, 10); // 20 (number)
```

Check out codetogo for [how to convert a string to a number](#).

Saving booleans

We run into a similar issue with booleans because the boolean `true` or `false` will be saved as a string, making it `"true"` or `"false"`.

```
const isVerified = true;

localStorage.setItem("isVerified", isVerified);
```

When you read it, it will be `"true"`, thus the easiest way to convert it back to a boolean is by comparing it to the string "true":

```
const isVerified = localStorage.getItem("isVerified") ===
"true";
```

With the code above, `isVerified` will only be `true` when `"true" === "true"`. So if localStorage contained `"false"`, the result of `"false" === "true"` will be false meaning `isVerified` will be false.

Note: we will remind you in the next chapter about how to convert to a number and boolean. In this chapter, we're only focusing on saving to localStorage.

To store the state in localStorage, you have to use the localStorage API **inside** a `useEffect` call:

```
import {useState, useEffect} from "react";

function App() {
    const [random, setRandom] = useState(Math.random());

    useEffect(() => {
        // every time the value of random changes, save it to
localStorage:
        localStorage.setItem("random", random);
    }, [random]);

    return <button onClick={() => setRandom(Math.random())}>Re-
render</button>;
}
```

Every time the state `random` changes, the effect will run, which will persist the new value in localStorage under the key `random`. In the next chapter, we'll learn how to restore that value and set it as a default value for the state.

Recap

- The localStorage API is to be used sparingly and inside `useEffect`.
- The localStorage API can only save *strings*. So the key and the value will be strings.
- `localStorage.setItem("key", "value")` is how you save a key/value pair to localStorage.

Was this helpful?

Storing arrays & objects

Last updated July 2022

The localStorage API only supports storing strings as keys and values, so when you want to store `arrays` or `objects`, you need to convert them to a string first.

You can do so with the `JSON.stringify()` function.

Remember from the [immutability chapter](#) that arrays and objects are both considered objects in JavaScript. So everything in this lesson will apply to both arrays and objects.

Convert an object to a string

```
const person = {
    id: 1,
    name: "Sam"
};

JSON.stringify(person); // '{"id":1,"name":"Sam"}'
```

Saving objects to localStorage

Once you convert the array or the object to a string, you can save it in localStorage:

```
const person = {
    id: 1,
    name: "Sam"
};

localStorage.setItem("person", JSON.stringify(person));
```

When you want to restore that object from localStorage, you have to convert it from a string back to an object/array, you can do so with `JSON.parse()`:

```
console.log(person); // {id: 1, name: "Sam"}
```

Recap

- Arrays & objects are both considered objects in JavaScript.
- When saving arrays/objects to localStorage, you need to convert them to a string first.
- `JSON.stringify(object)` will convert an object/array to a string.

Was this helpful?

Chapter recap

Last updated July 2022

We will see in the following chapter how we can restore the persisted values when the user returns to our application. This was the first half of the process. See you in the next chapter!

Chapter Recap

- The localStorage API is to be used sparingly and inside `useEffect`.
- The localStorage API can only save *strings*. So the key and the value will be strings.
- `localStorage.setItem("key", "value")` is how you save a key/value pair to localStorage.
- Arrays & objects are both considered objects in JavaScript.
- When saving arrays/objects to localStorage, you need to convert them to a string first.
- `JSON.stringify(object)` will convert an object/array to a string.

Was this helpful?

Restoring from localStorage

Last updated July 2022

In the previous chapter, we made it possible for our components to store their state in localStorage.
In this chapter, we'll learn how to restore the state from localStorage, meaning that the initial value we will give to `useState` will come from localStorage.

This will allow you to persist the dark mode setting, for example. So if the user chooses dark mode and comes back to your website, it will remain in dark mode because we've stored that value in localStorage.

Let's see how we can read the value stored in localStorage, as it is a bit different depending on the stored data type.

Strings

Reading strings from localStorage is the most straightforward option because localStorage stores everything as strings.

```
localStorage.getItem("key-here");
```

Numbers

Because localStorage will always give us a **string**, we need to manually convert it to a Number:

```
let value = localStorage.getItem("key-here");
value = Number.parseInt(value, 10); // convert to number
```

You can also use `Number.parseFloat(value, 10)` if you want to allow decimal points (numbers after the decimal, for example: `12.5`).

Booleans

Booleans could be either `true` or `false`. When saved to localStorage, they will be either `"true"` or `"false"` (because they get converted to strings).

The easiest way to convert those strings back to boolean is always to compare the localStorage entry to the string: `"true"`; here's why:

```
"true" === "true" // true (means original value was true)
"false" === "true" // false (means original value was false)
```

So here's how it would look like when reading from localStorage:

```
const value = localStorage.getItem("key-here") === "true";
```

By comparing it to the string `"true"`, you are converting it to a boolean.

Default value

When reading a key that doesn't exist (yet) from localStorage, it may be empty. In that case, it will return `null`.
You can default to a value in multiple ways. Using `if` conditions, the ternary operator, or the nullish coalescing operator (`??`). Here are some examples that show us how we can default to an empty array:

```
// using an if condition
let value = []; // default value
if (localStorage.getItem("key-here")) {
    value = localStorage.getItem("key-here");
}

// OR: using the ternary operator
const value = localStorage.getItem("key-here") !== null ?
localStorage.getItem("key-here") : [];
```

```
const value = localStorage.getItem("key-here") ?? [];
```

It's often important to default to a value if you're expecting an array. This will make sure that calling `.map()` or other array methods will not fail if `localStorage.getItem()` returns `null`.

How do we restore from localStorage?

Let's see that in the next lesson, we'll learn about *lazy initial state*.

Recap

- Reading strings from localStorage is the most straightforward option because localStorage stores everything as strings
- Numbers need to be converted from strings using `Number.parseInt("string", 10)`
- Booleans need to be compared with the string `"true"`, which will convert them to boolean.

Was this helpful?

Lazy initial state

Last updated April 2024

Theoretically speaking, restoring from localStorage is as easy as replacing the initial value in useState with a localStorage call, for example:

```
import {useState} from "react";

function App() {
    // do NOT use this (for performance reasons)
    const [state, setState] =
useState(localStorage.getItem("some-state"));
}
```

The code above works but creates a performance issue.

`localStorage.getItem("some-state")` is a relatively slow operation and we **only** need it for the initial state. However, every time the App component re-renders, we will end up running `localStorage.getItem("some-state")` even though we do NOT need that value anymore.

Lazy initial state

Lazy initial state is here to fix the issue. It allows you to pass a *function definition* rather than an expression that will allow React to only call it on the initial render.

Here's what it looks like:

```
import {useState} from "react";

function App() {
    // use this 👍
    const [state, setState] = useState(() ⇒
```

It is a *very subtle* adjustment, look at the before/after:

```
// before
const [state, setState] = useState(localStorage.getItem("some-state"));
// after
const [state, setState] = useState(() ⇒ localStorage.getItem("some-state"));
```

The difference is that **before** we had an expression: `localStorage.getItem("some-state")` and **after**, we have a function definition: `() ⇒ localStorage.getItem("some-state")`.

This will allow React to only call the function needed to get the value of the initial state **once**, which will not create performance issues.

So in comparison, the first (slow) approach is an expression that runs every time (because React cannot decide to skip an expression). In contrast, the second (fast) approach is a function that runs only the first time. When you provide a function definition, React has the ability to **only** run it the first time.

The function above is an arrow function with implicit return, but you can use any kind of function you want as long as it returns the value for the initial state. The above example can be re-written as:

```
// Also valid
const [state, setState] = useState(() ⇒ {
    return localStorage.getItem("some-state")
});
//Also valid
const [state, setState] = useState(function() {
    return localStorage.getItem("some-state")
});
```

Recap

- Lazy initial state allows you to avoid performance issues when the initial state requires a potentially expensive computation.
- Lazy initial state works by defining a function that will return the initial state rather than an expression. This allows React to call that function only once.

Was this helpful?

Restoring arrays & objects

Last updated February 2023

Remember that we can treat arrays & objects the same way and that we're *stringifying* (converting to string) arrays & objects before saving them to localStorage.

This means that when we want to read them, we need to `parse` them into objects again.

Here's how you can do that:

```
import {useState, useEffect} from "react";

function App() {
    const [array, setArray] = useState(() => {
        return JSON.parse(localStorage.getItem("key-here"));
    });

    useEffect(() => {
        localStorage.setItem("key-here", JSON.stringify(array));
    }, [array]);
}
```

It can also be rewritten using implicit return:

```
const [array, setArray] = useState(() =>
JSON.parse(localStorage.getItem("key-here")));
```

Recap

- When reading arrays & objects from localStorage, we need to parse them back into objects with `JSON.parse(string)`.

Chapter recap

Last updated September 2022

Great job on completing this chapter!

You're now ready for **Project VIII**. See you there!

Chapter Recap

- Reading strings from localStorage is the most straightforward option because localStorage stores everything as strings
- Numbers need to be converted from strings using `Number.parseInt("string", 10)`
- Booleans need to be compared with the string `"true"`, which will convert them to boolean.
- Lazy initial state allows you to avoid performance issues when the initial state requires a potentially expensive computation.
- Lazy initial state works by defining a function that will return the initial state rather than an expression. This allows React to call that function only once.
- When reading arrays & objects from localStorage, we need to parse them back into objects with `JSON.parse(string)`.

Was this helpful?

Persist Online Store Admin

Last updated July 2022

The next project is an opportunity for you to practice and recap the following topics:

- Effects
- Restoring from localStorage

Enjoy (and take your time)!

Was this helpful?

Quick Promises refresher

Last updated July 2022

Promises are a fundamental requirement for working with the Fetch API. They are a JavaScript language feature, thus independent from React.
A Promise is used whenever some piece of code completes in the future.

If you'd like a course to learn promises, consider **Learn JavaScript**.

Resolving a promise

The most common knowledge you need to have regarding promises is how to **resolve** them.
Resolving a promise means performing an action when the promise has completed its work.

Assuming the function called `functionThatReturnsPromise`, here's how you resolve it:

```
functionThatReturnsPromise().then(result => {
    console.log(result);
});
```

To resolve a Promise, you need to call `.then()` on it and pass a callback function. That callback function will be called when the promise successfully completes its work.

The first argument passed to that callback function (in this example, `result`) will be the data that the function has been computing. This is all theoretical because we're talking about promises in general. In the next lesson, we'll be working with the Fetch API and resolving its promises to clarify things.

Catching errors

called. You will need to have a `.catch()` call to catch errors. Here's an example:

```
functionThatReturnsPromise()
.then(result => {
    console.log(result); // when successful
})
.catch(error => {
    console.error(error); // when there's an error
})
```

We will have an entire chapter covering failed fetch calls.

Recap

- We will use promises when working with the Fetch API.
- To resolve a promise you need to call `.then(result => {})`
- To catch an error thrown by a promise you need to call `.catch(error => {})`

Was this helpful?

Fetch API

Last updated July 2022

The `fetch` API is a browser API (meaning that it's a functionality available in the browser) that lets you make a network request to another server.

The most common uses of `fetch` API when building (web) apps are sending and receiving data from the backend/API.

The `fetch` API returns a promise which is why we have to resolve it.

Here's an example of using the fetch API to **GET** the data from this backend/API: https://react-tutorial-demo.firebaseio.com/users.json.

```
fetch("https://react-tutorial-demo.firebaseio.com/users.json")
.then(response => response.json())
.then(data => {
    console.log(data);
});
```

There are four things to note here:

1. the `fetch()` call returns a promise, which is why we resolve it with `.then()`.
2. the `response` we get back from `fetch` is a generic response, which is why we want to convert it to a JSON object by calling `response.json()`.
3. `response.json()` also returns a promise, so we need to resolve it **again** with `.then()`.
4. always start with `console.log(data)` because every backend/API will return different data based on the purpose of that API.

The blueprint of fetch is the following:

```
fetch(URL)
.then(response => response.json())
.then(data => {
```

Feel free to add the code above to your notes so that you can reuse it whenever needed.

Do we need Axios?

It's common to see online tutorials recommending that you download a package called `axios` to use the `fetch` API. However, this is not necessary at all.
Using `axios` makes the fetch code above shorter by one line, however, throughout this course, we'll teach you how to write your own hook, called `useFetch`, which is also why you will NOT need to use `axios`.

Recap

- `fetch(url)` returns a promise that resolves to the **response**.
- The **response** is a generic response that needs to be converted to JSON with `response.json()`.
- `response.json()` also returns a promise that must be resolved.
- **Always** start with a `console.log(data)` because every backend/API returns different data.
- You do NOT need to use axios. The `fetch` API is powerful on its own and the `useFetch` hook will make it easier to write.

Was this helpful?

Chapter recap

Last updated July 2022

We saw in this chapter a quick overview of how to get data from a backend/API using the `fetch` API.

In the next chapter, we'll learn how to use the `fetch` API inside **React components**.

Chapter recap

- We will use promises when working with the Fetch API.
- To resolve a promise you need to call `.then(result => {})`
- To catch an error thrown by a promise you need to call `.catch(error => {})`
- `fetch(url)` returns a promise that resolves to the **response**.
- The **response** is a generic response that needs to be converted to JSON with `response.json()`.
- `response.json()` also returns a promise that must be resolved.
- **Always** start with a `console.log(data)` because every backend/API returns different data.
- You do NOT need to use axios. The `fetch` API is powerful on its own, and the `useFetch` hook will make it easier to write.

Was this helpful?

Fetch inside a Component

Last updated April 2024

Please read the note below that came into effect as of React 18.

As of React 18, it's not recommended to perform `fetch` operations inside `useEffect` as `useEffect` is now called *after* the component has rendered, which is a little bit too late for a fetch call that initializes the component with the data it needs.

While it's still technically possible to perform `fetch` in a React component using `useEffect`, React recommends using a library such as **useSWR**, **react-query**, or **React Router 6.4+'s data API**.

Suppose you're using a framework such as Next.js or Gatsby. In that case, you won't face these issues as data loading is baked into the framework itself.

There's another downside that the `useEffect` will run twice in Strict mode in development mode, making the duplicate `fetch` requests.

The benefit of having a library such as `useSWR` is that data fetching starts before rendering. Its result is also cached for a specific duration of time and can thus be reused. Also, they handle the double `useEffect` issue so that you don't see two `fetch` requests in development mode.

Here's a screencast on how to use `useSWR`:

Please check out the GitHub discussions for progress on the new React 19 version of this course.

We saw in the previous chapter how to use fetch to get some JSON data from a backend/API.
In this chapter, we'll use the fetch API inside a React component and create a state variable out of the data that we get back from the backend/API.

Where should you call fetch?

Calling fetch inside a component is considered a side effect because fetch goes to the network (outside of your component), so it must be inside a `useEffect`. `fetch` can be called in an event handler (for example, in response to a click). In that case, you must **not** wrap it with a `useEffect`. We will learn and practice both use cases.

The next question that comes to mind is, how often do you want to call that effect? The answer is that it depends, but most of the time, it will be just once when the component mounts. This means that we need to have the following useEffect:

```
useEffect(() => {
```

Fetching data

Now that we know **where** to call the fetch API inside our component, you can just use the fetch code from the previous chapter and put it inside the useEffect. Here's what it would look like:

```
import {useEffect} from "react";

function App() {
    useEffect(() => {
        fetch("https://react-tutorial-
demo.firebaseio.com/users.json")
        .then(response => response.json())
        .then(data => {
            console.log(data);
        });
    }, []);
}
```

This will run the fetch call **once** inside the component.

Linking it to state

How do we save that response into a state variable? The answer is in 2 steps:

1. Create a state variable. For example: `const [users, setUsers] = useState()`.
2. Call the setState function inside the `.then(data => {})`. For example: `.then(data => { setUsers(data) })`.

Here's how the full example looks like:

```
import {useEffect, useState} from "react";
```

```
    useEffect(() => {
        fetch("https://react-tutorial-
demo.firebaseio.com/users.json")
        .then(response => response.json())
        .then(data => {
            console.log(data); // keep it for debugging
            setUsers(data);
        });
    }, []);
}
```

It's important to remember that the `setUsers(data)` can **only** be called inside the `.then(data => {})` because before that or after that, the `data` variable is not defined.

Recap

- Calling fetch inside a React component is a side effect.
- Most of the time, fetch will need to be called once after the component has mounted. So the useEffect dependency will be `[]`
- Linking fetch to state: call the `setState` function inside the `.then(data => {})`.

Was this helpful?

Avoiding infinite loops

Last updated February 2023

The previous lesson showed us the starting point of using fetch inside React. Even though it looks like it's the complete solution, there are some common gotchas, edge cases, and other things you need to consider. We will cover these in the following chapters.

In this lesson, we'd like to focus on one of the most common gotchas: how to avoid creating infinite loops when using the fetch API.

The below example should NOT be used

```
import {useState, useEffect} from "react";

function App() {
  const [state, setState] = useState(Math.random());

  useEffect(() => {
    fetch("https://react-tutorial-
demo.firebaseio.com/users.json")
    .then(response => response.json())
    .then(data => {
      console.log(data);
      setState(data); // !!! ❌ this creates an infinite loop in
this example !!!
    });
  }); // !!! because the effect runs on every re-render !!!
}
```

The above example creates an infinite loop; here's why:

1. Component gets rendered
2. `useEffect` runs. Fetch API call to `users.json`.
3. The response arrives, and gets converted to JSON. We call `setState(data)`.

5. The same `useEffect` will run again. Fetch API call to `users.json`. And so on.

Your code will keep running the effect, calling setState, which ends up calling the effect again.

This can be solved by specifying `[]` as a dependency of the `useEffect`.

We will see in future lessons other examples where we need to specify another dependency for the `useEffect`.

Recap

- If you forget to specify the effect dependency and have a `setState` call inside the effect, you will end up with an infinite loop.

Was this helpful?

Chapter recap

Last updated March 2023

Great job on completing this chapter! The challenges are becoming more exciting as we use the fetch API.

Ignoring a fetch request in dev mode

The Beta React docs suggests creating an `ignore` boolean variable that you can use to *ignore* the duplicate fetch request.

We recommend using `useSWR` (or waiting for the new `use` hook) rather than wrapping all your fetch calls with this logic, as it will quickly get complicated.

For completeness' sake, please check out how to ignore the duplicate fetch request on the **React beta docs**.

In the next chapter, we'll learn how to show a loader/loading message while the fetch call is loading and cover another common gotcha when using fetch inside React.

Chapter Recap

- Calling fetch inside a React component is a side effect.
- Most of the time, fetch will need to be called once after the component has mounted. So the useEffect dependency will be `[]`
- Linking fetch to state: call the `setState` function inside the `.then(data => {})`.
- If you forget to specify the effect dependency, and you have a setState call inside the effect, you will end up with an infinite loop.

Cannot read property X of undefined

Last updated February 2023

Cannot read property X of undefined is one of the most common errors you get while working with the Fetch API in React.

Here's an example of a code that breaks with the following error:

```
import {useState, useEffect} from "react";

function App() {
    const [users, setUsers] = useState();

    useEffect(() => {
        fetch("https://react-tutorial-
demo.firebaseio.com/users.json")
        .then(response => response.json())
        .then(data => {
            console.log(data);
            setUsers(data);
        });
    }, []);

    // ❌ This will throw an error
    return <h1>There are {users.length} users</h1>
}
```

The code above will throw the error: **Cannot read property 'length' of undefined**.

This means that in the expression `users.length`, `users` has the value of `undefined`, which is why we can't access `.length` on it because that's like running `undefined.length`.

Why does this code break?

You have to remember that the `setUsers(data)` being called inside the `.then()` is running at some point in the future.

So the first time the App component renders, `users` is `undefined`, and we try to access `users.length`, which breaks. When the fetch resolves, and we get the data and call `setUsers(data)`, then `users.length` becomes **valid**. But, the component is already broken.

How do we solve it?

There are several ways to solve it depending on what the component is doing. Here are two approaches that you can use:

1. Conditional rendering

Conditional rendering is a topic covered in [chapter 19](). In short, you add an `if` condition and return a different JSX based on some condition.
In our example, we check `if (!users)` (or `if (users === undefined)`) and return `null` or a message for example `Loading ...` :

```
import {useState, useEffect} from "react";

function App() {
    const [users, setUsers] = useState();

    useEffect(() => {
        fetch("https://react-tutorial-
demo.firebaseio.com/users.json")
        .then(response => response.json())
        .then(data => {
            console.log(data);
            setUsers(data);
        });
    }, []);
```

```
    }

    return <h1>There are {users.length} users</h1>
}
```

2. Logical && operator

This will be explained in the next lesson!

Recap

- Trying to access the object/array returned from the fetch API before the fetch has resolved is a common error.
- Add an if condition and return a different JSX (conditional rendering) to avoid this issue.

Was this helpful?

Logical && operator

Last updated July 2022

Sometimes you don't want to render a completely different JSX but rather **skip** rendering the part of the JSX that depends on the `data` coming from the API. For example:

```
import {useState, useEffect} from "react";

function App() {
    const [users, setUsers] = useState();

    useEffect(() => {
        fetch("https://react-tutorial-
demo.firebaseio.com/users.json")
        .then(response => response.json())
        .then(data => {
            setUsers(data);
        });
    }, []);

    // This will throw an ERROR
    return <>
        <h1>Users</h1>
        <ul>
            {users.map(user => <li key={user.id}>{user.name}
</li>)}
        </ul>
  </>
}
```

The error will be `Cannot read property 'map' of undefined`.

So what if you want to suppress the error without changing the output of the component. So before the `users` have loaded, you just want it to render the following JSX:

```
    <h1>Users</h1>
    <ul>
    </ul>
</>
```

Notice how we skip the `users.map( ... )`.

And then, when `users` have loaded, the `users.map( ... )` will render.

This is a common scenario and can be easily solved using the [logical && operator](#).

You can prefix the `users.map()` call with `users &&` , the end result will look like this:

```
import {useState, useEffect} from "react";

function App() {
    const [users, setUsers] = useState();

    useEffect(() => {
        fetch("https://react-tutorial-
demo.firebaseio.com/users.json")
        .then(response => response.json())
        .then(data => {
            setUsers(data);
        });
    }, []);

    // this will work without breaking 👍
    return <>
        <h1>Users</h1>
        <ul>
            {users && users.map(user => <li key={user.id}>
{user.name}</li>)}
        </ul>
  </>
}
```

The reason why this works is that:

at users, meaning that it won't continue after the && because users returns undefined. This will prevent `users.map()` from running, which would have broken. So before `users` has loaded, the App component will render:

```
<>
    <h1>Users</h1>
    <ul>
    </ul>
</>
```

2. After 'users' has loaded: the expression `users && users.map()` will run in its entirety because `users` does not return a falsy value (such as `undefined` or `false`). Thus `users.map()` will be called.
So after `users` has loaded, the App component will render:

```
<>
    <h1>Users</h1>
    <ul>
        {users && users.map(user => <li key={user.id}>{user.name}
</li>)}
    </ul>
</>
```

Recap

- The logical && operator can be used to avoid rendering a specific part of JSX.
- This can be useful to prevent the component from breaking before the state has been set when the fetch response has been resolved.

Was this helpful?

Handling HTTP errors

Last updated July 2022

When working with the `fetch` API, some issues might occur with the fetch request. There are often two types of errors, HTTP errors and network errors.

In this lesson, we'll cover HTTP errors.

HTTP status codes

When you send a fetch request, you get a response with a status code.
A status code of 200 (or any number between 200 and 299, often written as 2xx) is considered successful.

More details about [HTTP status codes on MDN](#).

However, responses in the range of 400 to 499 (4xx) and 500 to 599 (5xx) mean that the fetch request was not completed successfully.
4xx refers to an issue relating to the **client**, which is you. This often means that you called the fetch request unexpectedly (for example, authentication is missing, a field is not provided, or the endpoint is not found).
On the other hand, 5xx errors refer to a server error, meaning that the issue is from the **server** (for example, the server cannot handle your request because it's overloaded with requests, or there's a bug in the server's code)

This lesson will teach us how to handle 4xx and 5xx errors while working with `fetch`.

Let's say you're fetching a certain endpoint that does not exist, here's what the code would look like:

```
fetch("https://react-tutorial-demo.firebaseio.com/validation-
example.json")
.then(response => response.json())
.then(data => {
```

What would happen is that instead of getting the `data` back, we get nothing. In some APIs, you might get a JSON object containing the error message and potentially an error code. There's no standard way; it depends on the API you're using.

However, as a general rule, the best thing to do here is to always check for the data you're getting before calling the setState on it. You can do that by wrapping it in an `if` statement.

Unfortunately, we can't give a recommendation that will work in all cases. This depends on the API, as they're quite different in error handling, but this is often mentioned in their documentation.

So here's an example of how you'd avoid setting state on an error object, assuming that the API returns an `error` object when things don't go as expected:

```
import {useEffect, useState} from "react";

function App() {
    const [users, setUsers] = useState();

    useEffect(() => {
        // this fetch request is designed to always fail
        fetch("https://react-tutorial-
demo.firebaseio.com/validation-example.json", {
            method: "post"
        })
        .then(response => response.json())
        .then(data => {
            console.log(data);
            // only set the users when there's no error
            if (data && !data.error) {
                setUsers(data);
            }
        });
```

Recap

- Depending on the API you're working with, you will most likely need to wrap the `setState` call with an **if** condition to avoid calling setState on an error object.

Was this helpful?

Handling network errors

Last updated July 2022

Network errors happen when the connection breaks or when the user is offline. In this case, the `fetch` call will **throw** instead of resolving. This means you can catch this error by adding a `.catch()` block, here's an example:

```
fetch("https://react-tutorial-demo.firebaseio.com/users.json")
.then(response => response.json())
.then(data => {
    console.log(data);
})
.catch(error => {
    console.log(error); // or console.error(error)
});
```

We added the `.catch` block above. If the internet connection breaks while this request is being sent or if the user is offline, then the `.then()` won't run, and the `.catch` will run instead, which will print the error message to the console.

You can then show a message to the user telling them that something went wrong and potentially offer the possibility to retry it.

Recap

- `.catch` allows you to handle network errors in fetch requests.

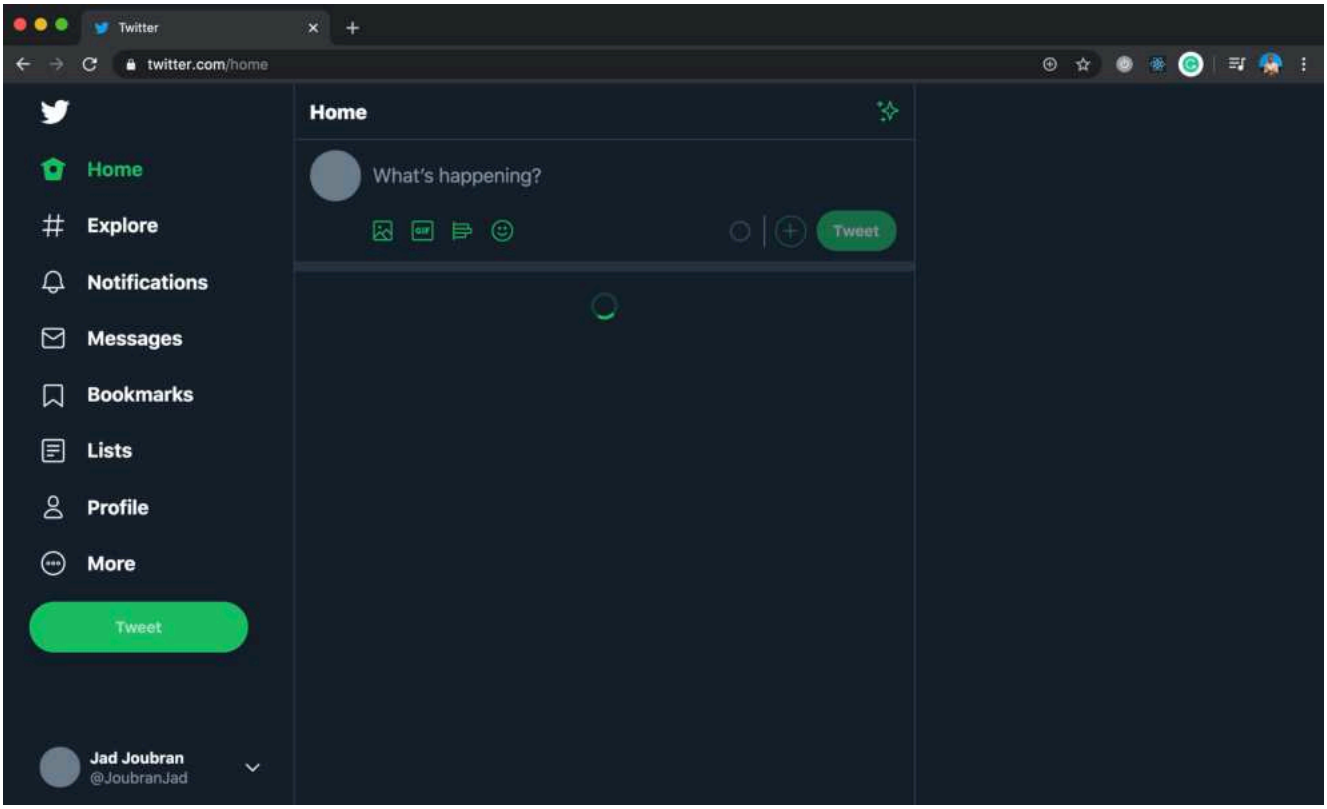
Was this helpful?

Handling fetch loading

Last updated February 2023

When fetching data from the network using the fetch API, it's often important to show a loader which is a visual indicator to the user that some data is being fetched from the network and that they should wait a moment.

It's **not always** recommended to have a loader. Still, as a general rule, if the main content where the user is looking is being loaded, it's a good idea to have a loader. Here's an example:



Keep in mind that you will most likely have a fast connection while developing your app on your computer. But remember that your users might have a slower connection, or they could be browsing on their phones. Mobile connections tend to be slower than Wi-Fi at home or work.

How to show a loader?

So we start by defining a state variable that will indicate whether the fetch request is loading or not; we'll call it `isLoading` and we'll make it default to `false`:

```
// inside a React component
const [isLoading, setIsLoading] = useState(false);
```

Start the loading

We need to `setIsLoading(true)` as soon as the fetch request starts, for example:

```
import {useEffect, useState} from "react";

function App() {
    const [isLoading, setIsLoading] = useState(false);

    useEffect(() => {
        setIsLoading(true); // start the loader
        fetch("https://react-tutorial-
demo.firebaseio.com/users.json")
        .then(response => response.json())
        .then(data => {
            console.log(data);
        });
    }, []);
}
```

In this case, since the component will run a fetch request as soon as possible, you can decide to change the `useState(false)` to `useState(true)`, then you won't need to start the loader before the `fetch` call. Here's how it looks:

```
import {useEffect, useState} from "react";

function App() {
    const [isLoading, setIsLoading] = useState(true); // create
```

```
    useEffect(() => {
        fetch("https://react-tutorial-
demo.firebaseio.com/users.json")
        .then(response => response.json())
        .then(data => {
            console.log(data);
        });
    }, []);
}
```

Stop the loading

The code above will set `isLoading` to `true` and will never stop that loader. This is why we need to stop the loader when we get a response.

However, we should also stop the loader in case the fetch request breaks, so we need to call `setIsLoading(false)` inside the `.then()` and the `.catch()`.

```
import {useEffect, useState} from "react";

function App() {
    const [isLoading, setIsLoading] = useState(true); // create
and start the loader

    useEffect(() => {
        fetch("https://react-tutorial-
demo.firebaseio.com/users.json")
        .then(response => response.json())
        .then(data => {
            setIsLoading(false); // stop the loader
            console.log(data);
        })
        .catch(error => {
            setIsLoading(false); // stop the loader
            console.error(error);
        });
    }, []);
}
```

Recap

- As a general rule, if the main content where the user is looking is being loaded, it's a good idea to have a loader.
- Loaders are important for users on slow connections & mobile connections.
- To show a loader, we need to have a state variable that keeps track of whether the request is loading or not. For example, a state variable called `isLoading`.
- To start the loader, we need to call `setIsLoading(true)` when the fetch request starts.
- To stop the loader, we need to call `setIsLoading(false)` when the fetch request resolves successfully or with an error.

Was this helpful?

Promise.finally

Last updated February 2023

You may have noticed the repetition of code in the previous examples where we had to call `setIsLoading(false)` in 2 places:

1. inside `.then()`
2. inside `.catch()`

There's a better alternative which is `.finally()`. This is a JavaScript feature for *Promises*, here's an example:

```
functionThatReturnsPromise().then(() => {
    console.log("success");
})
.catch(() => {
    console.log("error");
})
.finally(() => {
    console.log("done");
});
```

When the `functionThatReturnsPromise` resolves successfully, the console will log: `success` and `done`.
When it rejects with an error, the console will log: `error` and `done`.

Notice how the callback passed to `.finally` will run in both scenarios.

We can take advantage of that and call the `setIsLoading(false)` inside the `.finally`.

The code from the previous lesson becomes:

```
import {useEffect, useState} from "react";
```

```
and start the loader

    useEffect(() => {
        fetch("https://react-tutorial-
demo.firebaseio.com/users.json")
        .then(response => response.json())
        .then(data => {
            console.log(data);
        })
        .catch(error => {
            console.log(error);
        })
        .finally(() => {
            setIsLoading(false); // stop the loader
        });
    }, []);
}
```

Note: it's still important to **keep the `.catch`** block even when using `.finally`:

```
.catch(error => {
    console.log(error);
})
```

because if the fetch request fails and you don't have a `.catch` block, you will get an error that will break the whole component.

Use JSX to show a loader

There are a variety of ways for you to show a loader. Here's how you show a `Loading ...` message:

```
function App() {
    const [isLoading, setIsLoading] = useState(true);

    useEffect(() => {
```

```
    if (isLoading) {
        return <p>Loading ... </p>;
    }
}
```

You can also use the logical && operator or the [ternary operator](#) to render a loading message:

```
function App() {
    const [isLoading, setIsLoading] = useState(true);

    useEffect(() => {
        // same code above ...
    }, []);

    return (<>
        {isLoading && <p>Loading ... </p>}
        {/* OR */}
        {isLoading ? <p>Loading ... </p> : ""}
    </>);
}
```

Disable a button while loading

You can also disable a button while loading; this is common practice to prevent the user from repeatedly clicking on a button that has already started a fetch request. For that, you need to set the `disabled` attribute equal to the expression `{isLoading}`. For example:

```
<button disabled={isLoading}>Click me</button>
```

Recap

- You can call `setIsLoading(false)` inside `.finally()` instead of having to call it inside `.then()` and `.catch()`.
- Any method of conditional rendering will allow you to render a loading message while `isLoading` is `true`.
- You can disable a button while loading: `<button disabled={isLoading}>Click me</button>`.

Was this helpful?

Chapter recap

Last updated October 2023

It's important to pay attention to these small details because the User Experience of your web app is very important.

You can adopt the step-by-step approach I use when building products:

1. Make a basic version work
2. Use it as a user. Can I make it better?
3. Slow down the Internet connection using Dev Tools. Network throttling on **chrome** and **firefox**
4. Is it clear to the user that resources are being loaded from the network?
5. Can I make it even better?
6. Refactor code if necessary

Chapter Recap

- As a general rule, if the main content where the user is looking is being loaded, it's a good idea to have a loader.
- Loaders are important for users on slow connections & mobile connections.
- To show a loader, we need to have a state variable that keeps track of whether the request is loading or not. For example, a state variable called `isLoading`.
- To start the loader, we need to call `setIsLoading(true)` when the fetch request starts.
- To stop the loader, we need to call `setIsLoading(false)` when the fetch request resolves successfully or with an error.
- The callback passed to `.finally()` will run in both scenarios: success & error.
- You can call `setIsLoading(false)` inside `.finally()` instead of having to call it inside `.then()` and `.catch()`.

- You can disable a button while loading: `<button disabled={isLoading}>Click me</button>`.

Was this helpful?

Fetch onClick

Last updated July 2022

A common use case is for you to fetch some data when the user clicks on a button.

You can do so by writing the `fetch` code inside the `handleButtonClick` function, for example:

```
function App() {
    const [isLoading, setIsLoading] = useState(false);

    function handleButtonClick() {
        setIsLoading(true);
        // fetch("...") as usual
    }

    return <button onClick={handleButtonClick}>Load
data</button>;
}
```

So when you want to run fetch onClick, the fetch call would be inside the `handleButtonClick` function rather than in the `useEffect`.

Notice how we defaulted the `isLoading` to `false` because it will only start loading when the user clicks on the button. We start the loader inside `handleButtonClick` with `setIsLoading(true)`.

Disable a button while loading

As we saw in the previous chapter, if you keep an `isLoading` state variable, you can disable that button **while** it's loading.

```
<button disabled={isLoading}>Click me</button>
```

This lesson is less guided because you already know the concepts independently, and you'll get to practice that in the next challenge.

Recap

- You fetch data on click by placing the `fetch` call inside the `onClick` handler.
- You can disable a button while it's loading with `<button disabled={isLoading}>Click me</button>`.

Was this helpful?

Fetch onChange

Last updated September 2022

Another common use case is needing to fetch some data when the user selects a value from a dropdown.

This is very similar to the chapter about **Effect, State & Events**.

Remember how we had to wrap our `console.log(currency)` with a `useEffect` that has a dependency on `currency`. This allowed us to run a piece of code whenever `currency` changed.

```
import {useState, useEffect} from "react";

function CurrencySelector() {
    const [currency, setCurrency] = useState("");

    useEffect(() => {
        console.log(currency);
    }, [currency]);

    function handleCurrencyChange(event) {
        setCurrency(event.target.value);
    }

    return <>
        <h3>Select currency</h3>
        <select onChange={handleCurrencyChange}>
            <option value="usd">USD</option>
            <option value="eur">EUR</option>
        </select>
        <h1>{currency}</h1>
    </>;
}
```

Now do that inside the handleCurrency change because at that point in time, you won't have the most up-to-date value of `currency` (the state).

Instead the fetch call should be inside the:

```
useEffect(() => {
    console.log(currency);
    // fetch call here
}, [currency]);
```

Recap

- If you need to run a fetch call based on a state value that can change, it's best to run the fetch request inside a useEffect with a dependency on that state variable.

Was this helpful?

Fetch with async/await

Skip chapter?

Last updated February 2023

This chapter is **optional**. If you already know about `async/await` and would like to use it in React, then you can solve the following challenges using `async/await`.
If you're not a fan of `async/await`, then you can solve the challenges using traditional promises, like the previous chapters, and the tests will still pass.

That's because `async/await` is syntactic sugar on top of promises that allows you to express your promise logic as a set of operations that run line by line. But don't forget that async/await is still running actual promises and calling the `.then()` for you.

If you're not super familiar with `async/await`, we recommend you stick to promises just like the previous chapters.

Let's take the example back from [Chapter 50](#):

```
import {useEffect, useState} from "react";

function App() {
    const [users, setUsers] = useState();

    useEffect(() => {
        fetch("https://react-tutorial-
demo.firebaseio.com/users.json")
        .then(response => response.json())
        .then(data => {
            setUsers(data);
        });
    }, []);
}
```

If you'd like to use `async/await`, we need to prefix the function with the keyword `async` to be able to use the keyword `await`. **However**, an `async` function is a function that returns a Promise.

return a cleanup function.

So the code below does **NOT** work:

```
useEffect(async () => {
    // this will break
});
```

Instead, we need to keep the `useEffect` as a regular function and have a new function definition inside that gets called immediately (also called Immediately Invoked Function Expression):

```
useEffect(() => {
    (() => {
        // this is a function that will execute immediately
    })();
});
```

Let's break it down:

```
// 1) function definition:
() => {

}

// 2) wrap it with parentheses
(() => {

})

// 3) call this function with ()
(() => {

})();
```

passed to `useEffect`.

Now we need to make this new function an `async` function by prefixing it with the keyword `async`:

```
useEffect(() => {
    (async () => {
        // we can use await here ✅
    })();
});
```

And now finally, for every function that returns a promise (for example `fetch` and `response.json()`) we can prefix it with `await` and remove the `.then()`.

```
import {useEffect, useState} from "react";

function App() {
    const [users, setUsers] = useState();

    useEffect(() => {
        (async () => {
            const response = await fetch("https://react-tutorial-
demo.firebaseio.com/users.json")
            const data = await response.json()
            setUsers(data);
        })();
    }, []);
}
```

We can now read the code line by line.

Recap

- To use `await`, we need to be inside an `async` function.

because then it would return a Promise, whereas React is expecting either `undefined` or a cleanup function.

- Instead, we add an async immediately invoked function expression inside `useEffect`.

Was this helpful?

Fetch, events & async/await

Skip chapter?

Last updated August 2022

Luckily, async/await in events is easier than with the case of `useEffect`.

All you need to do is add `async` in front of the function and you're ready to use the `await` keyword. For example:

```
function App() {
    async function handleButtonClick() {
        // use fetch with await here
    }

    return <button onClick={handleButtonClick}>Load
data</button>;
}
```

Error handling

When using promises, we used to use `.catch` to handle errors. With `async/await`, you would have to wrap the `await` code with a try...catch block.

```
import {useEffect, useState} from "react";

function App() {
    const [users, setUsers] = useState();
    const [isLoading, setIsLoading] = useState(true);

    useEffect(() => {
        (async () => {
          try {
            const response = await fetch("https://react-tutorial-
demo.firebaseio.com/users.json")
            const data = await response.json();
```

```
            } catch (error) {
                console.log(error)
                setIsLoading(false);
            }
        })()
    }, []);

    return null
}
```

Notice how we stop the loader after `response.json()` in the `try` block and in case of an error in the `catch` block.

You can also use the `finally` keyword to refactor the common code between the `try` and `catch`:

```
import {useEffect, useState} from "react";

function App() {
    const [users, setUsers] = useState();
    const [isLoading, setIsLoading] = useState(true);

    useEffect(() => {
        (async () => {
            try {
                const response = await fetch("https://react-tutorial-
demo.firebaseio.com/users.json")
                const data = await response.json();
                setUsers(data);
            } catch (error) {
                console.log(error)
            } finally {
                setIsLoading(false)
            }
        })()
    }, []);
```

The `finally` block will execute in both scenarios, whether the `try` call was successful or not.

Recap

- For events, add `async` in front of the function, and you're ready to use the `await` keyword.
- To handle errors, you need to wrap your code with a try...catch block.

Was this helpful?

Chapter recap

Skip chapter?

Last updated January 2023

Awesome! It's time for Project IX.

After Project IX and X, we'll show you how to create your own hook that abstracts most of the code needed for `fetch`. So if you're not a fan of `async/await` or how lengthy the code has been, don't worry; we'll make it more elegant!

Chapter Recap

- To use `await`, we need to be inside an `async` function.
- You can't make the function passed to `useEffect` an `async` function because then it would return a Promise, whereas React is expecting either `undefined` or a cleanup function.
- Instead, we add an async immediately invoked function expression inside `useEffect`.
- For events, add `async` in front of the function, and you're ready to use the `await` keyword.
- To handle errors, you need to wrap your code with a `try...catch` block.

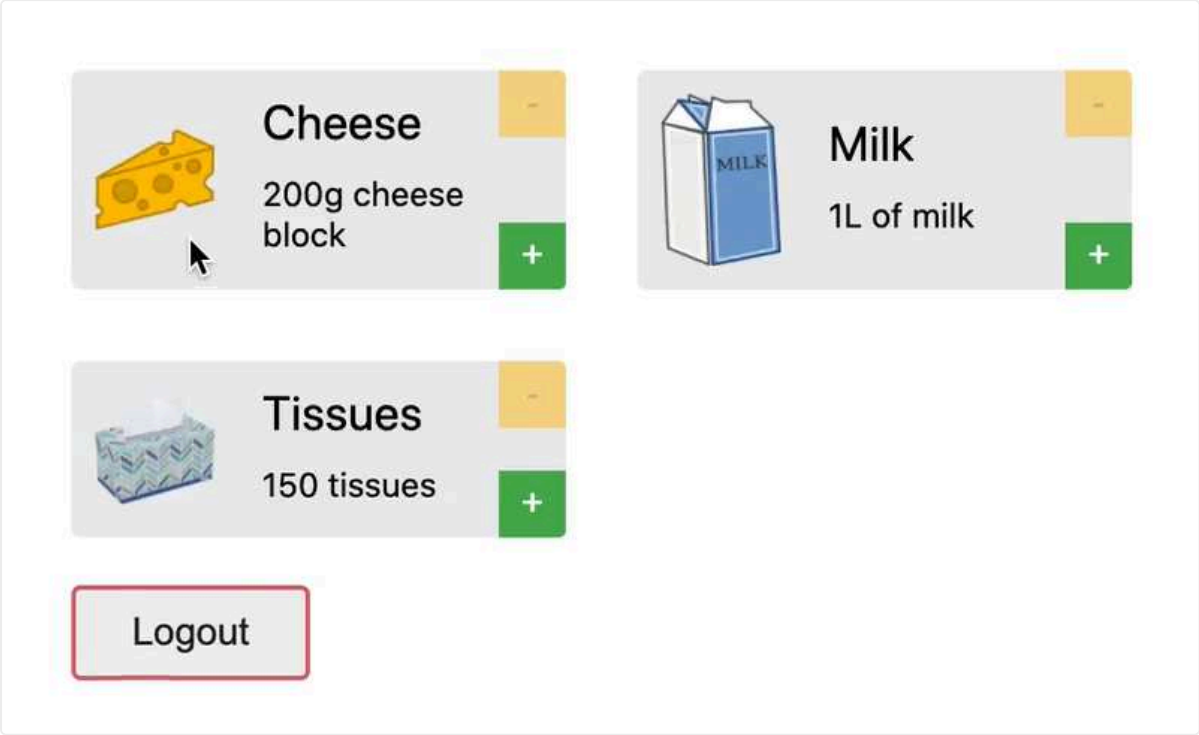
Was this helpful?

Project IX

Last updated July 2022

In this project, we will build on top of **Project III** and fetch the products from an API.

That's because when building a supermarket app, the products will most likely change; thus, they will be provided by an API.



Goal of Project IX

The goal of Project IX is to solidify your knowledge of using `fetch` inside React as well as the concepts we learned early on in the course:

- JSX
- State
- Effects
- Conditional rendering

All while building a real-life project.

Was this helpful?

Fetch POST

Last updated July 2022

In this lesson, we'll look at what `fetch` POST does, when it's used, and how it's used outside of React. In the next lesson, we'll see how it's used in React.

The `fetch` calls we've made in the last chapters were fetch GET calls. On the other hand, POST is often used to *send* information with the `fetch` request.
For example, when you're logging in, you send the `email` and the `password` as the body of the `fetch` call.

The fetch request looks the same as before, except for the second argument, where you need to specify the **method**.

So with `fetch` GET, we used to write:

```
fetch("https://react-tutorial-demo.firebaseio.com/grades.json")
.then(response => response.json())
.then(data => {
    console.log(data);
});
```

We don't need to specify the second argument because the `method` defaults to `GET`. So the above code is equivalent to the following:

```
fetch("https://react-tutorial-demo.firebaseio.com/grades.json",
{
    method: "GET" // this is a default
})
.then(response => response.json())
.then(data => {
    console.log(data);
});
```

```
fetch("https://react-tutorial-demo.firebaseio.com/grades.json",
{
    method: "POST"
})
.then(response => response.json())
.then(data => {
    console.log(data);
});
```

This will allow us to make a POST request.

However, most of the time, you'd need to send some data with the request. In this example, we need to send the `grade`. To do that, you need to add to the second argument the following key and value `body: JSON.stringify({grade: 10})` (or any grade between 0 and 100).

We wrap the `{grade: 10}` with a JSON.stringify call which will convert the object into a string.

Another thing we recommend you always do when working with APIs that return data in JSON is to specify the following header: `Content-Type: "application/json"`.

Think of it as if you were telling the API: I am sending you some data in JSON format.

Putting it all together, here's how our fetch POST looks like:

```
fetch("https://react-tutorial-demo.firebaseio.com/grades.json",
{
    method: "POST",
    headers: {
        "Content-Type": "application/json"
    },
    body: JSON.stringify({grade: 50})
})
.then(response => response.json())
.then(data => {
```

Feel free to try the code above, which will add the grade (it has to be a number between 0 and 100). And then open this link https://react-tutorial-demo.firebaseio.com/grades.json, and you will see your `grade` added to the list.

It is lengthy to make a `fetch` POST request, but don't worry, as we'll handle that once we learn how to create our own hooks ;)

Recap

- `fetch` POST is used to send data to an API.
- `fetch` POST is similar to `fetch` GET, except that you have to specify the second argument of the `fetch()` function.
- You need to specify the `method: "POST"` and often have to send the `body: JSON.stringify(dataObjectHere)`.
- It's also a best practice to set the header `"Content-Type": "application/json"`.

Was this helpful?

Chapter recap

Last updated July 2022

The next chapter is **Project X**, where you will be given the opportunity to work with forms again while also working with `fetch`.

Here's what you need to remember from this chapter:

```
fetch(ENDPOINTURL, {
    method: "POST",
    headers: {
        "Content-Type": "application/json"
    },
    body: JSON.stringify(DATAOBJECT)
})
.then(response => response.json())
.then(data => {
    console.log(data);
});
```

This is a boilerplate that you can use whenever working with `fetch` POST. Make sure to replace the `ENDPOINTURL` and the `DATAOBJECT`.

Chapter Recap

- `fetch` POST is used to send data to an API.
- `fetch` POST is similar to `fetch` GET, except that you have to specify the second argument of the `fetch()` function.
- You need to specify the `method: "POST"` and often have to send the `body: JSON.stringify(dataObjectHere)`.
- It's also a best practice to set the header `"Content-Type": "application/json"`.

Send one of the state variables in the body?

Was this helpful?

Introduction to custom hooks

Last updated July 2022

Throughout this course, we've seen the following hooks provided by React:

- `useState`
- `useEffect`

There are a couple more provided by React. However, what's interesting is that you are able to build your **own hooks**.

Benefits

Building your own hooks allows you to:

1. easily reuse code in several components, which makes it easier to share common functionality.
2. build abstractions that make some complicated logic easier to read.

We will see both of these use cases in the following few chapters. We'll reuse the **map** logic in several components, as well as build an abstraction on top of `fetch` to make it easier to use.

How does it work?

A custom hook is a JavaScript function whose name starts with **use**. This function is able to call other hooks (for example, `useEffect`, `useState`, etc.).

In this chapter and the next one, we're only going to use `useEffect` and `useState` in order to simplify the concept and teach it step by step.

My first custom hook

First we define the custom hook (remember to start its name with use):

```
function useHelloWorld() {
    console.log("Hello World!");
}
```

and then you can use it in any component (you will have to import, more on that later on):

```
function App() {
    useHelloWorld();
}
```

Every time the App component renders, you will see `Hello World!` in the console.

Not a useful example, but this is the simplest.
In the next lesson, we'll take it a step further!

Recap

- Building your own hooks allows you to:
- 1. easily reuse code in several components, which makes it easier to share common functionality.
- 2. build abstractions that make some complicated logic easier to read.
- A custom hook is a JavaScript function whose name starts with **use**.
- A custom hook is able to call other hooks (for example, `useEffect`, `useState`, etc.).

Was this helpful?

Custom hook with useEffect

Last updated July 2022

Let's take a look at an example where we build our own hook that uses `useEffect`:

```
import {useEffect} from "react";

function useHelloWorld() {
    useEffect(() => {
        console.log("Hello World!");
    });
}
```

You use this custom hook the same way we did in the previous lesson.

This will end up logging `Hello World!` to the console **after** every re-render (because the console.log is wrapped in useEffect).

You could also decide to run this only once when the component gets rendered the first time; in that case, you have to specify the dependency of useEffect as an empty array:

```
import {useEffect} from "react";

function useHelloWorld() {
    useEffect(() => {
        console.log("Hello World!");
    }, []);
}
```

When would this be useful?

This could be useful for sending an analytics event (for example, using Google Analytics).

```
import {useEffect} from "react";

function useAnalyticsEvent() {
    useEffect(() => {
        gtag("event", "component_rendered");
    }, []);
}
```

The `gtag` function is provided by the Google Analytics library (assuming you have it included in this project). The point of this example is to show you that you can send the analytics event only once after the component has rendered the first time. That's because we're using `useEffect( ... , [])`.

You can do so much more with custom hooks, such as receiving parameters, returning items, calling `useState`, etc.

We're going to see all of these capabilities throughout the following chapters.

Recap

- Your custom hook can call `useEffect` as long as `useEffect` is imported.
- Everything you know from `useEffect` still applies (for example, dependencies array).

Was this helpful?

Moving hooks into their own file

Last updated July 2022

The goal of custom hooks is to be able to reuse them in several components. It's also much cleaner to have them in a separate file. That's precisely what we're going to do in this lesson.

There is no specific recommendation as to what to call the file, so we've come up with our own. Please note that not everyone follows this because it's a personal recommendation. Still, you are encouraged to use it in your projects.

To make things cleaner, name the file that contains a custom hook: `[useSomething].js`, replacing the `[useSomething]` with the name of your hook.

For example, coming back to this custom hook:

```
import {useEffect} from "react";

function useHelloWorld() {
    useEffect(() => {
        console.log("Hello World!");
    });
}
```

It's recommended that you call it `useHelloWorld.js`.

Now when you put this function in its own file, you need to **export** it so that you're able to use it in other files.
Just like we did when working with **multiple components**, you need to `export default` this function. So our custom hook becomes:

```
// useHelloWorld.js
import {useEffect} from "react";

export default function useHelloWorld() {
```

```
    });
}
```

Two things to note here:

1. Don't forget to import `useEffect` in this file because we're using it.
2. `export default` the custom hook function so we can import it.

Now back in our App component, we can import it:

```
import useHelloWorld from "./useHelloWorld.js";

function App() {
    useHelloWorld();

    return <h1>My App</h1>;
}
```

This is much cleaner now as we have the hook in its own file, and we're able to import it and use it in any other component.

You can also decide to name the file `useHelloWorld.hook.js` to make it even clearer that this is a custom hook. It all comes down to personal preferences.

Recap

- The recommendation of this course is to name your hooks `[useSomething].js`, replacing the `[useSomething]` with the name of your hook.
- Don't forget to import `useEffect` (or other React hooks) that you're using in the file where you define your custom hook.
- You need to `export default` your custom hook so that you can import it in other files.

Custom hooks with parameters

Last updated February 2023

Since custom hooks are just functions, they can accept `parameters`. You can provide one or more parameters, of any data type, just like any other function.

In this chapter, we're going to start with a basic example where we only provide one parameter, which is of type string.

Let's take a look at this example:

```
function useWelcomeGreeting(name) {
    useEffect(() => {
        console.log(`Welcome ${name} to my app`);
    }, []);
}
```

This hook accepts the `name` of the user, which you can specify when calling the hook (when calling the function `useWelcomeGreeting`). Here's how you can use it:

```
// assuming you imported that function (and it was exported)

function App() {
    useWelcomeGreeting("Sam");
}
```

When the App component renders the first time, it will log `Welcome Sam to my app`. Not the most useful example, but you will build a useful one in the next challenge ;)

Remember the rules of hooks

Back in Chapter 21, we learned about the **Rules of hooks**. We had two rules:

We actually over-simplified that rule back then and said that you should only call hooks inside **components**. However, it **is** possible to call Hooks from your own custom hooks, which we already did in the previous chapter and this one.
For example, you can call `useEffect` inside your own custom hook.

So Rule #1 is: Only call Hooks from React functions, for example, **components** or **custom hooks**. But you cannot call React hooks in any other function or place.

Rule #2: Only call Hooks at the Top Level

This rule still applies! When you use your own custom hook, make sure you don't call it inside loops, conditions, or nested functions because the order of the hooks has to stay the same between re-renders.

useDocumentTitle

In the next challenge, you will write your custom hook called `useDocumentTitle`. We'll give you a couple of hints, but try to start thinking about it already ;)

Here's how you'll end up using it:

```
// You also need to import useDocumentTitle.

function App() {
    useDocumentTitle("Welcome to the online store")
}
```

When the `App` component renders, it will change the `document.title` to **Welcome to the online store**.

Now look at this other example:

```
import {useState} from "react";
// You also need to import useDocumentTitle ...
```

```
    const [count, setCount] = useState(0);
    useDocumentTitle(`${count} products in your shopping list`);

    function handleButtonClick() {
        setCount(prevCount => prevCount + 1);
    }

    return <button onClick={handleButtonClick}>Add item</button>
}
```

If you pay attention to how you code the `useDocumentTitle` custom hook, then every time you click on the button, you should end up with the updated title (with the new count).

You will be able to try that in this chapter!

Recap

- Since custom hooks are just functions, they can accept `parameters`. You can provide one or more parameters, of any data type, just like any other function.
- The rules of hooks still apply to custom hooks.
- Rule #1: Only call Hooks from React functions (components or custom hooks).
- Rule #2: Only call Hooks at the Top Level

Was this helpful?

Refactor map integration

Last updated July 2022

In this project, you will refactor the Mapbox map integration from **Project VI**. Feel free to keep a browser tab of that project open so you can use your access token and other Mapbox configuration.

You will refactor the Mapbox map integration by creating a new custom hook called `useMapboxMap()`.

The benefit of having the map integration as a custom hook is that you can easily reuse it in several places on your website without having to repeat the access token and other boilerplate code.

The custom hook should be called `useMapboxMap(container)`. It should accept **one parameter**, the `container` where the map will be rendered. This will be a string representing the `id` of the element in the DOM.
Note: as mentioned before, we will soon learn about **refs**, so the `container` logic will be refactored later in this course.

Was this helpful?

Chapter recap

Last updated July 2022

Nice work!

You can take this idea a step further and make the `center` and `zoom` level customizable too.

With the below custom hook:

```
import {useEffect} from "react";

export default function useMapboxMap(container, center, zoom) {
    mapboxgl.accessToken =
"pk.eyXXXXX_REPLACE_WITH_YOUR_OWN_KEY";

    useEffect(() => {
        const map = new mapboxgl.Map({
            container: container,
            style: 'mapbox://styles/mapbox/dark-v10',
            center:  center,
            zoom: zoom
        });
    }, []);
}
```

You will be able to dynamically set the `center` and `zoom` level as follows:

```
useMapboxMap("map", [ 12.567898, 55.67583 ], 9);
```

You can also have a default configuration for the map and allow customizing all of its options:

```
import {useEffect} from "react";
```

```
"pk.eyXXXXX_REPLACE_WITH_YOUR_OWN_KEY";

    const defaultOptions = {
        container: "map",
        style: 'mapbox://styles/mapbox/dark-v10',
        center:  [ 12.567898, 55.67583 ],
        zoom: 9
    };

    useEffect(() => {
        const map = new mapboxgl.Map({
            ... defaultOptions,
            ... options
        });
    }, []);
}
```

By spreading both objects and having the `defaultOptions` spread first, you are able to **default** the values to the `defaultOptions` but then override any of the values with the `options` object. This is called object concatenation. For example:

```
useMapboxMap({
    zoom: 10
})
```

will end up rendering the map in the `container: "map"` (from defaultOptions), with the `center: [ 12.567898, 55.67583 ]` (from defaultOptions) but with a `zoom` of 10.

That's because:

```
{
    ... defaultOptions,
    ... {
        zoom: 10
```

will end up with the following object:

```
{
    container: "map",
    style: "mapbox://styles/mapbox/dark-v10",
    center: [
        12.567898,
        55.67583
    ],
    zoom: 10
}
```

Feel free to review **object concatenation**.

Was this helpful?

Custom hooks with state

Last updated June 2024

Just like how custom hooks can contain `useEffect` calls, custom hooks can also contain `useState` calls.

So you can create state variables inside your custom hooks, allowing you to refactor common logic further.

For example, for our supermarket web app, we will often need to have a **counter** that keeps track of how many items the user has added to their basket. This counter cannot go below 0 because you cannot order -1 items. Such a counter would look like the following:

```
import {useState} from "react";

function App() {
    const [counter, setCounter] = useState(0);

    function handleIncrementClick() {
        setCounter(prevCounter => prevCounter + 1);
    }

    function handleDecrementClick() {
        setCounter(prevCounter => {
            // only decrement if it's bigger than 0
            if (prevCounter > 0) {
                return prevCounter - 1;
            }
            // don't let it go below 0
            return 0;
        });
    }
}
```

```
// useProductCounter.js
import {useState} from "react";

export default function useProductCounter() {
    const [counter, setCounter] = useState(0);

    function increment() {
        setCounter(prevCounter => prevCounter + 1);
    }

    function decrement() {
        setCounter(prevCounter => {
            if (prevCounter > 0) {
                return prevCounter - 1;
            }
            return 0;
        });
    }

    return ; /* we need to return the counter and the two
functions */
}
```

Notice how we move the `useState` declaration and the two functions and place them into our new custom hook. We also renamed `handleIncrementClick` to `increment` because this is a reusable `increment` function, and it doesn't *necessarily* need to be called on click. Same thing for the `handleDecrementClick` function, which is now called `decrement`.

Now this custom hook needs to expose the **counter** state as well as the `increment` and `decrement` functions, so that they can be used by the outside component.

There are two main approaches to do this:

1. Array destructuring

array will have three items:

```
return [counter, increment, decrement];
```

Then inside your component, you would use array destructuring:

```
const [counter, increment, decrement] = useProductCounter();
```

This works because we returned an array of 3 items; the first corresponds to the `counter`, the second corresponds to the `increment` function, and the third corresponds to the `decrement` function.

This works, but as soon as you start having more than two items, it often feels cleaner to follow the second approach. But remember this is often dictated by personal style rather than strict rules.

2. Object destructuring

The second approach is to return an object containing the `count` and the two functions:

```
return {counter, increment, decrement};
```

This is using the ES2015 object shorthand syntax. It's exactly the same as the code below, but just shorter:

```
return {
    counter: counter,
    increment: increment,
    decrement: decrement
};
```

But because the keys and values have the same names, you can just write it as `return {counter, increment, decrement}`.

similar to array destructuring but for objects.

```
const {counter, increment, decrement} = useProductCounter();
```

Notice how the only change is that we changed the `[]` from the array to `{}` because `useProductCounter()` is now returning an *object*.

This approach is recommended when you have more than two items returned by a custom hook because the order won't matter if you use the same keys' names. For example, the example below is still valid because we have the same key names, and the order doesn't matter, which makes your code less error-prone:

```
const {increment, decrement, counter} = useProductCounter();
```

Updating the event listener

Now that our custom hook is returning the `increment` and `decrement` functions, we need to update the JSX's `onClick` handlers to use the new `increment` and `decrement` functions:

```
function App() {
    const {counter, increment, decrement} = useProductCounter();

     return <>
        <h2>{counter}</h2>
        <button onClick={increment}>+</button>
        <button onClick={decrement}>-</button>
    </>;
}
```

Notice how the App component is much smaller as most of its functionality has been refactored into the `useProductCounter` custom hook.

Recap

- Custom hooks can contain `useState` calls.
- Your custom hook can return an array of data (for example, state variable, as well as functions to update that state variable)
- Your custom hook can return an object of data
- We recommend returning an object rather than an array when you have more than two items being returned.

Was this helpful?

Custom useFetch hook

Last updated July 2022

You may have noticed that the `fetch` code is often lengthy, which makes your component a bit more complicated than it should be. That's one of the main reasons why we're going to create a custom `useFetch` hook that will simplify our `fetch` logic. This will also help us avoid common mistakes when working with fetch, as all the `fetch` logic will be in the `useFetch` hook.

Starting with "get"

The `useFetch` hook will return a `get` function and a `post` function. We'll start with the `get` function.

The `get` function will allow us to make `get` requests. Once the custom `useFetch` hook is built, here's how you will be able to use it:

Instead of writing the below:

```
useEffect(() => {
    fetch("https://react-tutorial-
demo.firebaseio.com/users.json")
    .then(response => response.json())
    .then(data => {
        console.log(data);
    })
    .catch(error => console.log(error));
}, []);
```

You will be able to write the following:

```
const {get} = useFetch("https://react-tutorial-
demo.firebaseio.com/");
```

```
    get("users.json")
    .then(data => {
        console.log(data);
    })
    .catch(error => console.log(error));
}, []);
```

Notice how:

1. the `useFetch` function returns an object with the `get` function.
   We use *object destructuring* to get access to the `get` function.
2. the `useFetch` function accepts the `baseUrl`, which allows us to call `get("users.json")` later on without repeating the lengthy base URL.
3. `get("users.json")` automatically converts the response to JSON. We don't have to do that anymore as we can immediately resolve the promise and read the `data`.

In the next challenge, we'll guide you into writing a basic version of the `useFetch` hook (only the `get` function for now).

After this chapter, we'll give you a more advanced and complete version of the `useFetch` hook, which you can use in your projects.

Recap

- The custom `useFetch` hook will make our fetch logic easier.
- The custom `useFetch` hook will return a `get` and a `post` function, using object destructuring.
- The custom `useFetch` hook will accept the `baseUrl` as the only parameter.
- The `get` function will accept the `url` that you want to fetch, which gets added to the `baseUrl`.

Was this helpful?

UseFetch hook: post

Last updated July 2022

In this lesson, we'll continue writing our `useFetch` hook by adding a `post` method.

The `post` method works similarly to the get but should also accept the body being sent in the fetch request.

Instead of the code below:

```
 fetch("https://api.learnjavascript.online/demo/react/grades",
{
    method: "POST",
    headers: {
        "Content-Type": "application/json"
    },
    body: JSON.stringify({grade: 20})
})
.then(response => response.json())
.then(data => {
    console.log(data);
});
```

you will be able to write the following:

```
const {post} =
useFetch("https://api.learnjavascript.online/demo/react/");

post("grades", {
    grade: 20
})
.then(data => {
    console.log(data);
})
.catch(error => console.log(error));
```

needs to be sent with the post request.

The `fetch` request should also set the `method` to `"post"` as well as set the headers.

Recap

- The `post` function will accept the `url` that you want to fetch, which gets added to the `baseUrl` and will make a **post** fetch request.
- The `post` function will also accept the `body` that will be JSON stringified and sent with the fetch request.

Was this helpful?

Chapter recap

Last updated March 2023

Nice work! Below you can find a more advanced version of the `useFetch` hook that you can use in your projects.

> As mentioned in **chapter 49**, we recommend using `useSWR`, or React Query as of React 18 or waiting for the new `use` hook.

This version of the custom hook returns an object containing three items:

- the `get` function.
- the `post` function.
- the `loading` state, which will be a boolean.

This will allow you to manage the loading state in your component easily.

We will work with this custom `useFetch` hook in the following two chapters.

```
import { useState } from "react";

export default function useFetch(baseUrl) {
  const [loading, setLoading] = useState(true);

  function get(url) {
    return new Promise((resolve, reject) => {
      fetch(baseUrl + url)
        .then(response => response.json())
        .then(data => {
          if (!data) {
            setLoading(false);
            return reject(data);
          }
          setLoading(false);
          resolve(data);
        })
```

```
            reject(error);
          });
      });
    }

    function post(url, body) {
      return new Promise((resolve, reject) => {
        fetch(baseUrl + url, {
            method: "post",
            headers: {
              "Content-Type": "application/json",
            },
            body: JSON.stringify(body)
        })
          .then(response => response.json())
          .then(data => {
            if (!data) {
              setLoading(false);
              return reject(data);
            }
            setLoading(false);
            resolve(data);
          })
          .catch(error => {
            setLoading(false);
            reject(error);
          });
      });
    }

    return { get, post, loading };
  };
```

Refactoring to useFetch

Last updated July 2022

This chapter aims to use the existing `useFetch` hook to simplify the `fetch` logic in your components!

As a reminder, the `useFetch` hook returns an object from which you can destructure the `get` or `post` function depending on whether you need to perform a `get` request or a `post` request.

Additionally, you can also destructure the `loading` state variable, which is a boolean representing whether the `fetch` is in a loading state or not. For example:

```
import {useEffect} from "react";
import useFetch from "./useFetch.js";

function App() {
    const {get, loading} = useFetch("https://react-tutorial-
demo.firebaseio.com/");

    useEffect(() => {
        get("users.json").then(data => {
            console.log(data);
        })
        .catch(error => console.error(error));
    }, []);

    return (<>
        <h2>{loading ? "Loading..." : ""}</h2>
    </>);
}
```

Introduction to refs

Last updated July 2022

Let's say you've got the following component, and you'd like to *focus* the text field when the component renders the first time.

```
import {useEffect} from "react";

function App() {
    useEffect(() => {
        // how do you focus the input?
    }, []);

    return <input type="text" />;
}
```

Note: The **.focus()** method on `<input>` elements is a method that places the user's cursor inside the text field. For example, when you open https://google.com, the search box is automatically focused.

You might be tempted to use `document.querySelector()` to find the `<input type="text" />` but this is a bad idea for several reasons:

- It may not work (if the input has not been rendered yet, remember React creates React elements that make up the Virtual DOM and are later rendered to the browser's real DOM).
- Even if it works, you may get unexpected results as you continue developing your application. For example, if you re-use that component several times in your app.

The solution for that is to use React's **refs**. A *ref* allows you to hold a *reference* to a rendered DOM element. Having created this ref and assigned it to a React element in JSX, you will be able to call imperative methods on the rendered DOM element, such as the `.focus()`.

build an entire project without using (or with minimal usage) refs, and that's okay.

Here's how it works:

```
import {useRef} from "react";

function App() {
    const inputRef = useRef();

    return <input ref={inputRef} type="text" />;
}
```

The code will be explained step by step in the next lesson, but for now, we want to show you the end result before explaining it.

Having assigned the `inputRef` to the `<input />` using the `ref={inputRef}` we will be able to access the rendered DOM element with `inputRef.current`.

Then whenever you want to focus the input element, you can call `inputRef.current.focus()`. So the end result looks like this:

```
import {useRef, useEffect} from "react";

function App() {
    const inputRef = useRef();

    useEffect(() => {
        inputRef.current.focus();
    }, []);

    return <input ref={inputRef} type="text" />;
}
```

In the next lesson, we'll explain the code above in detail as well as cover the use cases for refs.

Recap

- `document.querySelector()` should never be used inside React as it could have unexpected results.
- Use refs as a last resort to find a specific element rendered to the DOM with React to perform an operation on it.

Was this helpful?

Using refs

Last updated July 2022

In the previous lesson, we saw the code below that focuses the `<input />` element when the App component is rendered.

```
import {useRef, useEffect} from "react";

function App() {
    const inputRef = useRef();

    useEffect(() => {
        inputRef.current.focus();
    }, []);

    return <input ref={inputRef} type="text" />;
}
```

These are the 3 steps to create ref:

1. Import the `useRef` hook (named export) from the `react` package. `import {useRef} from "react";`.
2. Create a new ref at the top of the component (rules of hooks). For example, `const inputRef = useRef();`.
3. Assign the created ref to the JSX element that you're rendering, using the `ref` attribute. For example, `<input ref={inputRef} />`.

We recommend you always end the ref's variable name with the word `Ref`. So, for example:

- `inputRef` when referencing an `input`
- `buttonRef` when referencing a `button`
- `carouselRef` when referencing a `carousel` (a plugin, for example)

When you call `useRef()`, React will create an object containing the key `current`. So calling `const inputRef = useRef();` will return an object that looks like this:

```
{
    current: undefined
}
```

Then when you assign the ref to the JSX element, React will assign the `current` key to the rendered DOM element. So, for example:

```
{
    current: <input type="text" />
}
```

This means that every time you want to access the element you're referencing, you have to use the `.current` key. This is why in our example above, we had to call `inputRef.current.focus()`. That's because `inputRef.current` will allow you to access the referenced DOM element.

Timing issues

You have to keep in mind that `inputRef.current` will only return the DOM element **after** the component has been rendered to the DOM.
This is why you will often only use the ref inside an event handler or a `useEffect` call. That's because both of these methods run **after** the component has been rendered to the DOM.

When to use refs?

You might be tempted to use refs in every component, but in fact, refs are not commonly used, which is why they were left until this chapter. There are specific use

- Focusing an element.
- Playing a video (`videoElement.play()`), pausing a video, etc.
- Selecting text from the DOM.
- Initializing DOM libraries (will be covered in the next project).

The use cases can be categorized into two categories:

1. Imperative DOM operations. When you find that the only way to make this work, is by calling a certain DOM method. For example `.play()` or `.focus()`.
2. Initializing DOM libraries. For example, MapBox, or a carousel plugin.

Recap

- When you create a ref, React creates an object that looks like: `{current: undefined}`.
- When you assign the ref to a JSX element, you will be able to access the rendered element with `ref.current`.
- The DOM element will only be available inside an event handler or `useEffect` because you have to wait for the component to be rendered before it becomes available.
- The use cases for refs are:
- 1. Imperative DOM operations.
- 2. Initializing DOM libraries.

Was this helpful?

Refs in DOM libraries

Last updated July 2022

Back in **Project XI**, we created a custom hook that initializes and renders the MapBox map.

However, that custom hook relied on the `id` of the div that will contain the map. It was the `<div id="map"></div>`.

Instead of relying on the `id`, we should use a `ref`.

So in this project, you will rewrite the App component such that you pass the `ref` rather than the `id` of the map container.

Note that Mapbox allows you to pass either the `id` or the result of `document.querySelector()`, which is why we are able to pass a string or a ref. Also, even though we're going to be using *refs*, we still keep the `id="map"` on that div because it has some styling in `index.css`, which sets the map height.

Was this helpful?

Introduction to Context

Last updated July 2022

The word context in programming can be defined as *relevant information*. That's also the usage of Context in React. It's for when you want to make some information relevant to your app available for several components.

Before we define what Context is in React and how you can use it, let's start by exploring a problem.

Global data

Let's say your app is available in dark theme and light theme. This can be represented with a state variable in the topmost component, the App component. The `theme` will be defined as a state variable in the App component like so:

```
import {useState} from "react";

function App() {
    const [theme, setTheme] = useState("dark");

    return <Nav />
}

function Nav() {
    return <Button>Login</Button>;
}

function Button(props) {
    return <button>{props.children}</button>;
}
```

However, we need to know the current `theme` in the `Button` component because we'll be rendering a different class based on the theme. Thus, we'll have to pass down the

```
import React, {useState} from "react";

function App() {
    const [theme, setTheme] = useState("dark");

    return <Nav theme={theme} />
}

function Nav(props) {
    return <Button theme={props.theme}>Login</Button>;
}

function Button(props) {
    // render class depending on props.theme
    return <button>{props.children}</button>;
}
```

Notice how every component in our app will need to receive the `theme` as a prop. This will make your components bloated with code.

Use Context

This is a great example of where we can use Context. The ideal use case for Context is when you want to make global data accessible to several components in your app. For example, the theme of your app, the locale (language selected), the chosen currency, or the user profile details (id, image, bio). The settings that will reflect on **several** components in your UI.

Instead of having to pass down this data throughout all of your React component tree, we'll use Context!

Recap

- The word context in programming can be defined as *relevant information*.

- The ideal use case for Context is when you want to make global data accessible to several components in your app.
- Use cases for using Context: theme or locale (language selected), currency, user profile details (id, image, bio).

Was this helpful?

Create Context

Last updated September 2022

We will split the concept of implementing contexts in React into two parts:
Part 1: Creating the context Part 2: Using the context (from a component)

In this lesson, we'll cover the first part, how to create a context. Let's break that down step by step:

Step 1: Create Context

The first step is to create a new **Context**. For that, we need to import the named export: `createContext` from the React package `import {createContext} from "react"`.
Then we need to call the `createContext()` function and store it in a variable.

```
import React, {createContext} from "react";

const ThemeContext = createContext();
```

Step 2: Create Provider

Every context needs a Provider. A provider will provide the data to the context. In our case, we're creating a `ThemeContext`, and we'd like to provide the `theme` (for example `"dark"` or `"light"`).

```
import { createContext } from "react";

const ThemeContext = createContext();

function ThemeProvider(props) {
  const theme = "dark";

  return (
```

```
    </ThemeContext.Provider>
  );
};
```

Notice how the `ThemeProvider` is a React component that receives `props` and renders the `props.children` wrapped with `<ThemeContext.Provider>`.
The `ThemeContext.Provider` receives a `value` which is the value that all the children will be able to receive. This will become clearer once we follow the next lesson and use the context that we defined.

The `.` syntax is similar to the `React.StrictMode` syntax. `ThemeContext` is an object and `Provider` is a key on that object. We'll have a dedicated lesson for creating similar JSX objects in the following JSX Advanced chapter.

In a nutshell, this Provider will wrap our `<App />` component and give it the `value="dark"` so that we can read the `theme` from any component.

Step 3: Export the Context & the Provider

Finally, we need to export the `ThemeContext` and the `ThemeProvider`: `export {ThemeContext, ThemeProvider}`.

Putting it all together

Here's the complete code:

```
import { createContext } from "react";

const ThemeContext = createContext();

function ThemeProvider(props) {
  const theme = "dark";

  return (
    <ThemeContext.Provider value={theme}>
```

```
  );
};

export { ThemeContext, ThemeProvider };
```

Recap

- Steps for creating a context:
- 1. Create the context
- 2. Create the provider for that context
- 3. Export the context & the provider.
- The provider will wrap our components and provide them with the value (some global data).

Was this helpful?

Using Context

Last updated July 2022

Now that we have defined the `ThemeContext` and `ThemeProvider`, we're going to use them in our app.

Step 1. Wrap components with the Provider

To make the context available in your app, you have to wrap the components with the Provider.

Let's say we've got the App component from this chapter that renders the `<Nav />`:

```
// index.js
function App() {
    return <Nav />;
}
```

First, we need to import the `ThemeProvider` and wrap all our components with it:

```
// index.js
import {ThemeProvider} from "./ThemeContext.js";

function App() {
    return (<ThemeProvider>
        <Nav />
    </ThemeProvider>);
}
```

This will allow all the components nested inside `ThemeProvider` to have access to the context provided by `ThemeProvider`.

Step 2. Use the context inside components

Any component nested inside the ThemeProvider can now access the context by importing the `ThemeContext` and passing it to the `useContext()` hook. Here's how we do it:

```
// Button.js
import {useContext} from "react";
import {ThemeContext} from "./ThemeContext.js";

function Button(props) {
    const theme = useContext(ThemeContext);
    console.log(theme); // "dark"
    return <button>{props.children}</button>;
}
```

Notice how we import the `useContext` hook and we pass to it the `ThemeContext`. This will allow you to consume the `value` that we passed to the provider in the previous step, which in this example is `"dark"`.

This most likely seems over-complicated and convoluted, but remember that the benefit of this is that we don't have to pass the `props.theme` from the top-most component down to all the children component.

So we can refactor our code from the first lesson of this chapter into the following:

```
import {useContext} from "react";
import {ThemeContext, ThemeProvider} from "./ThemeContext.js";

function App() {
    return (<ThemeProvider>
        <Nav />
    </ThemeProvider>);
}

function Nav() {
    return <Button>Login</Button>;
}
```

```
    console.log(theme); // "dark"
    return <button>{props.children}</button>;
}
```

Recap

- To use the context you need to:
- 1. Wrap your components with the Provider.
- 2. In component files, pass the Context to the `useContext` hook.
- The benefit of using context is that we don't have to pass down the same prop throughout the entire component tree.

Was this helpful?

Context value

Last updated July 2022

In the previous chapter, we've always defined the context to return a string value:

```
function ThemeProvider(props) {
  const theme = "dark";

  return (
    <ThemeContext.Provider value={theme}> /* value is a string
(in this example) */
      {props.children}
    </ThemeContext.Provider>
  );
};
```

However, you can also decide to return an array or an object!

The benefit of returning an object is that you can return several values simultaneously. That might be useful, for example, if you're passing user details (such as first name, last name, etc.).

It's also helpful because it lets you pass a function as one of the entries in that object. For example, we can pass the following object:

```
{
    theme: "dark",
    toggleTheme: function() {
        // a function that toggles the theme
    }
}
```

Here's how you'd create such context:

```
import { createContext } from "react";
```

```
function ThemeProvider(props) {
  const theme = "dark";

  function toggleTheme() {
    // toggle the theme here
  }

  const value = {
    theme: theme,
    toggleTheme: toggleTheme
  }

  return (
    <ThemeContext.Provider value={value}>
      {props.children}
    </ThemeContext.Provider>
  );
};

export { ThemeContext, ThemeProvider };
```

Notice how the `<ThemeContext.Provider />` still accepts the same `value={}`, but instead of passing a string, we're now passing an object.

In the next lesson, we'll implement the `toggleTheme` function and show you how to use that from one of the components of your app.

Recap

- The context provider's value does not have to be a string. It can be an array or an object.
- The benefit of returning an object is that you can return several values simultaneously.
- Another benefit of returning an object is that you can return a function as one of the entries of that object.

function to toggle that theme.

Was this helpful?

Update context value

Last updated July 2022

Since we want to be able to change the value of the `theme` at some point, we need to change the theme from a variable into a state variable.

We can indeed use the `useState` hook inside the `Context`:

```
import { createContext, useState } from "react";

const ThemeContext = createContext();

function ThemeProvider(props) {
  const [theme, setTheme] = useState("dark");

  function toggleTheme() {
    // toggle the theme here
  }

  const value = {
    theme: theme,
    toggleTheme: toggleTheme
  }

  return (
    <ThemeContext.Provider value={value}>
      {props.children}
    </ThemeContext.Provider>
  );
};

export { ThemeContext, ThemeProvider };
```

And so now we can implement the `toggleTheme` function:

```
function toggleTheme() {
```

```
    } else {
      setTheme("dark")
    }
}
```

which is toggling the `theme` from `"dark"` to `"light"` and vice versa.

With that context we can now update the `theme` from anywhere in the app, as long as we use this `ThemeContext`. Here's how:

```
import {useContext} from "react";
import {ThemeContext} from "./ThemeContext.js";

function App() {
    const context = useContext(ThemeContext);
    console.log(context); // {theme: "dark", toggleTheme:
Function}

    return <button onClick={() => context.toggleTheme()}>Toggle
Theme</button>;
}
```

Notice how `useContext(ThemeContext)` is now returning an object containing the `theme` and the `toggleTheme` function.
We then call the `context.toggleTheme()` function when the button gets clicked (`onClick`).

Re-rendering

The state update caused by `context.toggleTheme()` will end up notifying (re-rendering) all the components that use this ThemeContext which is why you will immediately see the new `theme` being rendered.

Was this helpful?

Project XIV

Last updated July 2022

Here's a preview of what you'll build:

A common bug that developers encounter when working with context happens in the following scenario:

```
import {useContext} from "react";
import {createRoot} from "react-dom/client";
import {SomeContext, SomeProvider} from "./SomeContext.js";
import Blog from "./Blog.js";

function App() {
    const context = useContext(SomeContext);
    console.log(context); // undefined (this does NOT work) 👎

    return <SomeProvider><Blog />></SomeProvider>
}

createRoot(document.querySelector("#react-root")).render(<App
/>);
```

provider, you will be able to access the value of the context because the current component is NOT wrapped by that provider. However, all the children of that component will be able to access the context (for example, `<Blog />`).

There are mostly two ways to solve this:

1. Update the render call

We don't recommend this approach for the scalability of your app. Eventually, you may have more than one Provider that you'd like to wrap around your `<App />` so this could become cumbersome:

```
createRoot(document.querySelector("#react-root")).render(<SomeProvider><App />
</SomeProvider>);
```

This works because the provider now wraps the `<App />` component, so the `useContext(SomeContext)` would return the context value.

2. AppWrapper

We recommend this approach as it lets you scale and has several components (for example, `react-router` later on) wrapping your `<App />`.

You can create a component called `AppWrapper` (or whatever you want), where you wrap your `<App />` component with the provider. Here's the full solution:

```
import React, {useContext} from "react";
import {createRoot} from "react-dom/client";
import {SomeContext, SomeProvider} from "./SomeContext.js";
import Blog from "./Blog.js";

function App() {
    const context = useContext(SomeContext);
    console.log(context); // this works now 👍

    return <Blog />
```

```
function AppWrapper() {
    // Wrap the <App /> component with the provider
    return (<SomeProvider>
        <App />
    </SomeProvider>);
}


createRoot(document.querySelector("#react-
root")).render(<AppWrapper />);
```

Notice how we're rendering the `<AppWrapper />`, which renders the `<App />` component after being wrapped with all the necessary components.

You can also decide to move that AppWrapper into its own file; it's up to you.

Was this helpful?

Dot Notation

Last updated November 2023

Since JSX is always converting your `<Component />` into a `React.createElement(Component, {})` call, it is possible to have an object that contains several components, for example:

```
const Buttons = {
    Default: function(props) {
        return <button className="btn-default">{props.children}
</button>;
    },
    Outline: function(props) {
        return <button className="btn-outline">{props.children}
</button>;
    }
}
```

Then you can use these components as follows:

```
<Buttons.Default>Login</Buttons.Default>
<Buttons.Outline>Register</Buttons.Outline>
```

The reason why this works is that they will be converted to React.createElement calls as follows:

```
React.createElement(Buttons.Default, {}, "Login");
React.createElement(Buttons.Outline, {}, "Register");
```

Usage

We've used that back in the Context chapters (`ThemeContext.Provider`):

```
      {props.children}
    </ThemeContext.Provider>
  );
```

It's also sometimes used in UI Libraries where the library would export all the buttons under a single object called `Buttons`, and then you can choose the type of button by using `Buttons.Default` or `Buttons.Outline`, for example.

This is also the same syntax that we used for `<React.StrictMode />`.

Recap

- `<Buttons.Default />` gets converted to `React.createElement(Buttons.Default, {})`

Was this helpful?

Dangerously set inner HTML

Last updated February 2024

Let's take the following component:

```
function Footer() {
    const text = "<strong>All rights reserved &copy;</strong>";

    return <div>{text}</div>;
}
```

You would expect to read: `All rights reserved` © (in bold); however, we end up seeing: `<strong>All rights reserved ©</strong>` in regular font-weight, and the copyright sign is showing as an HTML entity.

This is because whenever you have an expression in JSX, React will **escape** all the HTML entities. This is a security feature that prevents XSS injection.

What is XSS Injection?

Cross-Site Scripting (XSS) is a security vulnerability when the user is able to inject your page with their own code (HTML, CSS, or JavaScript). For example, imagine you would like to wish your friend a happy birthday on their social network. Instead of writing: `Happy Birthday`, what if you write:

```
<script>alert("You got hacked!")</script>
```

If the social network allows you to write your own scripts that get rendered on the page, this will be a massive security breach as you will be able to read people's cookies, act on their behalf, redirect them to other websites, and do so many other things.

So the "comment" above becomes: `<script>alert("You got hacked!")</script>`. By escaping the < and > characters, the browser is able to render the comment as text rather than a script.

JSX Expressions are escaped

Expressions in React are automatically escaped to prevent common XSS injections. This is great and should often remain the case.

However, there are minor scenarios where you would like to override this behavior and skip this prevention. For example, when you are 100% sure that the expression you've embedded in your JSX is coming from you (or your team) but never from the end-user. Here's how you can do that:

```
function Footer() {
    const text = "<strong>All rights reserved &copy;</strong>";

    return <div dangerouslySetInnerHTML={{__html: text}}></div>;
}
```

Notice how we removed the `{text}` expression and instead added a prop called `dangerouslySetInnerHTML`, which accepts an object `{__html: text}` which is wrapped with an expression `{}` (that's why we have `{{}}`, one for the object and one for the expression).
Inside this object, you have to set the key **__html** (that's two underscores). It is complicated, on purpose, to remind you that this is dangerous.

The golden rule is to **never trust user input**. If the expression you're embedding is coming from the end-user (for example, a comment, a first name, last name, email, etc.), you should **never** dangerously set inner HTML.

Here's a real-life example of when it makes sense to use `dangerouslySetInnerHTML`:

sometimes have HTML formatting (for example, bolding and all). These questions are 100% authored by me and never by the end-user, meaning it is an input I can trust. So to be able to render the HTML formatting, the component that renders the question uses `dangerouslySetInnerHTML`.

Recap

- Cross-Site Scripting (XSS) is a security vulnerability when the user is able to inject your page with their own code (HTML, CSS, or JavaScript).
- JSX expressions are always escaped.
- You should never use `dangerouslySetInnerHTML` if the content comes from the end-user. Never trust user input.

Was this helpful?

Multiple exports

Last updated July 2022

Back in the **components chapter, one component per file lesson**, we recommended having one component per file, which is true in most scenarios.

However, when building a UI Kit, you may find yourself exporting more than 1 component from the same file. Here's an example:

```
// Button.js
import clsx from "clsx";

function ButtonPrimary(props) {
    const {children, className, ...rest} = props;
    const classes = clsx("btn-primary", className);
    return <button className={classes} {...rest}>{children}
</button>;
}

function ButtonSecondary(props) {
    const {children, className, ...rest} = props;
    const classes = clsx("btn-secondary", className);
    return <button className={classes} {...rest}>{children}
</button>;
}

export {ButtonPrimary, ButtonSecondary};
```

Notice how we export two components in this example: `ButtonPrimary` and `ButtonSecondary`.
These are **named** exports rather than a *default* export. Here's how you import them from another file:

```
import {ButtonPrimary, ButtonSecondary} from "./Button.js";
```

Recap

- Generally, you will often export one (default) component per file. Still, when a certain file exports components closely related to each other, it's okay to export more than one (named) component.

Was this helpful?

Styles in React

Last updated March 2023

There are more than a dozen ways of writing CSS in React applications.

This is a topic that cannot be covered without giving personal preference, which means they're not bulletproof as well as plenty of developers will disagree with it. It's also quite tough to give guidance on such a topic without knowing the scope of the project that's going to be built with React.

Having said that, our recommendation is the following:

Stick to writing regular CSS, the way we've been doing it so far in this course, as much as possible. Unless:

- You are working in a team, and that team is using a certain tool for CSS.
- You are working in a very large team. In that case, they will already be using a certain tool for CSS.
- You already have experience and are comfortable with a certain tool for CSS.

We mean the "regular way of writing CSS," as creating CSS files where you create classes and then using those classes in the `className` prop.

Examples of tools

Some of the most popular tools are:

- **CSS Modules**. It allows you to import your styles as an object of class names which you can then use in your React component.
- **styled-components**. It uses a technique called CSS-in-JS. Where the CSS is written in JavaScript.

> While this is opinionated advice, we recommend avoiding `styled-components` as it may negatively impact performance.

We recommend using a UI library such as **ionic** (used for the flashcards app of this course), **material-ui**, or **antd** as this will make you write way less CSS. Alternatively, you can take a look at the promising **tailwind CSS**.

Generally speaking, it's always better to keep things you can solve with CSS written only in CSS.

Here's a video example of how Dark Mode can be implemented mostly using CSS with very minimal JavaScript:

youtube.com/watch?v=x4Xn0RvJEjI

Note that the course does give the example of context for dark mode management; that's because it is quite common in the industry, and we'd like you to be familiar with creating a dark mode using context too.

Some students have asked for additional advice on this topic. You can find that in this 👥 GitHub discussions thread.

Recap

- We recommend, in most cases, that you keep writing CSS in a separate CSS file and adding the class name in the `className` prop.

Was this helpful?

Deploying to Netlify

Last updated May 2023

This chapter covers deploying your application so that you can send it to your friends or even go live with your application with your own custom domain.

We recommend **Netlify.com** as a fast and convenient hosting platform for static projects with a generous free plan.

One-time setup

New site from Git

After creating an account on Netlify, you can click on **New site from Git** where you can deploy a site from your GitHub, GitLab, or Bitbucket repository.

Continue reading the below before clicking on that button. Netlify makes it easier by asking you (and attempting to pre-fill the correct answers) about the deploy settings when you create a new site from Git.

If you have already created your website, you can follow the instructions below to set the correct **Deploy settings**.

Deploy settings

Your React project contains your source files, so when we deploy the website, we should not serve the website from the root folder; but instead, it should be from the **build** folder.
Back in the **CRA/Vite chapter**, we talked about the `build` folder that gets created by running the `npm run build`. The `build` folder contains an optimized version of your app running using **React production**.

So we need to instruct Netlify about two things:

So from your Netlify site dashboard, you can click on **Deploys** (top menu), then **Deploy settings**, and then **Edit settings**.

In the **Build command**, you should specify: `npm run build` And the publish directory should be set to: `build`

By setting the build command, every time that Netlify needs to deploy your website, it will run `npm run build`.
This command will generate the optimized version of your website in the `build` folder, which will then be served to your users.

Netlify will provide you with a link where you can preview your website. In the next lesson, we'll talk about adding your own custom domain.

Auto-deploy

Netlify will automatically deploy your website whenever you push a new commit to the main branch.

If that's something you don't want, you can disable that feature in the same form above by choosing **Stop builds** instead of **Activate builds**.

This is all a one-time setup, meaning that afterward, you can enjoy your live website without reconfiguring the settings above!

If you don't feel comfortable about deploying your own app yet, don't worry! We'll link you back to this lesson in the Final project's recap.

Recap

- When deploying your website to Netlify, you need to change the **build settings** to instruct Netlify:

2. Which folder to publish: `build`

Was this helpful?

Custom domain

Last updated July 2022

If you'd like a custom domain for your website, you can also do so from Netlify.

From your site dashboard page, you can click on **Domain settings** and enter the domain you'd like.
Now one of the two scenarios below will occur:

1. Domain already has an owner

If the owner is not you, then you can't do anything. You have to choose a new domain. If you are the owner of that domain, you will then need to point your DNS settings to Netlify as instructed on that page.

Remember that changing DNS might take anywhere from a few minutes to a few hours. And it's always safer to wait 24 to 48 hours before going live worldwide with a new website so that DNS has had time to propagate worldwide.

2. Domain is available

If the domain is available, you can purchase it through Netlify (or another domain name provider).

If you choose Netlify, you won't have to mess with DNS configurations as it will automatically be configured.

Recap

- You can add a custom domain to your website through the **Domain settings** page in Netlify.

react-router-dom

Last updated December 2022

As we further develop our supermarket app, we find the need to add multiple pages. We've previously achieved that back in **Project III** where we conditionally rendered components based on whether the user is logged in or not.

But there are downsides to this approach:

- It will quickly get complicated if we want to add more pages.
- The page URL never changes. So if the user browses our web app and then decides to save the link or share it with a friend, it'll stay the same.

React Router solves these issues by letting us declaratively define routes that are also linked to a certain page URL. We will explain the expression "declaratively define routes" in the next lesson.

React Router

React Router is a library that makes creating routes in React easier. It is **not** built by the React team, thus it is **not** an official React package. However, it is very popular in the React community.

The name of the package is `react-router-dom` and it weighs 52kb minified.

The `react-router-dom` package uses *named exports* to export its functionality. This means that you will use the following syntax to import the components provided by this package:

```
import {} from "react-router-dom";
```

The name of the component will go in between the `{}`, that's because they're named exports.

following syntax:

```
import {BrowserRouter, Route} from "react-router-dom";
```

This course teaches the latest version of react-router-dom (v6).

The `react-router-dom` package exports many components and hooks. However, there are 10 of those that you will often use. Throughout the following few chapters, we will teach you these 10 components and hooks in this course. We will take the same step-by-step approach that you're used to by now.

For now, we want you to know that the 4 most common components that you will be importing in every app that uses React Router, are:

- `BrowserRouter`
- `Routes`
- `Route` (*Route* – singular of *Routes*)
- `Link`

We will explain these 4 components in the next lesson.

Visualizing the URL

In React Tutorial, you can visualize the current **URL** in the **Browser** tab, at the top. This URL starts with `http://localhost` and will then show your current **route**.
Note that this URL is only visible on challenges that use React Router.

Recap

- React Router allows us to declaratively define routes which will update the browser URL as we navigate through the app.
- React Router is not an official React library. But it's very popular in the React community.

- The `react-router-dom` package exports the following *named* exports: `BrowserRouter`, `Route`, `Routes`, and `Link`.

Was this helpful?

Basic navigation

Last updated March 2023

In the following lessons, we'll break down basic navigation between two pages, similar to the previous challenge where you visualized React Router for the first time.

There are four main concepts to explain:

1. `BrowserRouter`
2. `Routes`
3. `Route`
4. `Link`

1. BrowserRouter

The `<BrowserRouter />` component is a router that uses the HTML5 history API (mainly the **pushState** method that allows changing the URL of the page without triggering a browser refresh).
The BrowserRouter keeps your User Interface in sync with the URL, which means that the URL will always reflect the current route that is rendered.

There are other types of Routers but you will almost never need to use them. `BrowserRouter` is the most common one.
For the BrowserRouter to work, you have to *wrap* your entire application with that component:

```
import {BrowserRouter} from "react-router-dom";

function App() {
    return <BrowserRouter>
        <div>The rest of your app goes here</div>
    </BrowserRouter>;
}
```

application. In the following sections, you will see that the `<Routes />`, `<Route />` and `<Link />` will all go **inside** this `<BrowserRouter></BrowserRouter>`.

2. Route

The `<Route />` component will conditionally render a component when the `path` prop matches the current URL in the browser.

Here's an example of the `<Route />` component:

```
<Route path="/about" element={<About />}></Route>
```

The `<About />` component, provided to `Route` as an `element` prop, will **only** render when the browser URL matches the path `/about`.

Let's take a look at a full example:

```
import {BrowserRouter, Route} from "react-router-dom";

function App() {
    return <BrowserRouter>
        <div>The rest of your app goes here</div>

        {/*We still need to wrap these two Route components with
a <Routes />. Keep reading for now.*/}
        <Route path="/" element={<Home />}></Route>
        <Route path="/about" element={<About />}></Route>
    </BrowserRouter>;
}
```

When the user navigates to `/`, the `<Home />` component will be rendered.
Similarly, when the user navigates to `/about`, the `<About />` component will be rendered.
So the browser URL is matched against the `path` prop of every `<Route />`. This will determine the routes that will be visible.

Recap

- The `<BrowserRouter />` component is a router that uses the HTML5 history API (using the `pushState` method).
- You have to *wrap* your entire application with the `<BrowserRouter />` component.
- The `<Route />` component will conditionally render a component when the `path` prop matches the current URL in the browser.

Was this helpful?

Basic navigation (continued)

Last updated December 2022

3. Routes

The <Routes /> component will wrap several <Route /> components and will render the <Route /> that matches the current browser URL.

```
import {BrowserRouter, Routes, Route} from "react-router-dom";

function App() {
    return <BrowserRouter>
        <div>The rest of your app goes here</div>

        <Routes>
            <Route path="/" element={<Home />}></Route>
            <Route path="/about" element={<About />}></Route>
        </Routes>
    </BrowserRouter>;
}
```

Note that it is important that the <Routes /> element directly wraps the <Route /> elements, which means that the <Route /> has to be a direct descendant of the <Routes />. That means that you should not have a or other elements in between the <Routes /> and the <Route />; otherwise, the code won't work.

Declarative routing

This example over here:

```
<Routes>
    <Route path="/" element={<Home />}></Route>
```

is what we meant by React Router allows you to **declaratively define routes**. There are no `if` conditions.

You declare the routes in your application and provide a `path` for every route. And React Router will know which one to render.

Recap

- The `<Routes />` component will wrap several `<Route />` components and will render only the **first** `<Route />` that matches the current browser URL.
- The `<Routes />` element should directly wrap the `<Route />` elements.
- Declarative routing means defining the routes without `if` conditions. By providing the `path`, React Router will know which Route to render.

Was this helpful?

Route Link

Last updated December 2022

In the last two lessons, we explained the following components:

- `BrowserRouter`
- `Routes`
- `Route`

We still have to explain the `<Link />` component, which is a simple one.

We're used to link to a new page using the `anchor` tag. For example, `<a href="/about">About us</a>`. However, this link will perform a page navigation, which means our React app would have to restart. That's because the browser thinks we are navigating to a completely new page, so it performs an HTTP request to `/about`, and then the browser has to parse the HTML and the scripts, etc.

The benefit of React Router is that we can have instant navigation. That's why we need to replace the `<a>` link with the `<Link />` component provided by React Router. Here's how:

```
import {BrowserRouter, Routes, Route, Link} from "react-router-dom";

function App() {
  return (
    <BrowserRouter>
        <nav>
          <ul>
            <li>
              <Link to="/">Home</Link>
            </li>
            <li>
              <Link to="/contact">Contact</Link>
            </li>
          </ul>
```

```
        {/* Routes and Route goes here*/}
      </BrowserRouter>
    );
  }
```

The `<Link />` component takes a `to` prop. When the user clicks on the link, React Router will navigate to the path provided by the `to` prop.

This `<Link />` element will render an `<a>` element (for accessibility reasons explained below), but with some event handlers that will instruct React Router to do an instant redirect instead of a page navigation. So React Router is performing an `event.preventDefault()` to prevent the page from navigating to a new page. You don't have to do that; it's all declarative!

Accessibility

It's important to remember accessibility when relying so much on JavaScript. The `<Link />` element is accessible by default; that's because it ends up rendering an anchor tag `<a>`.
This will allow users that make use of a screen reader to know that this is a link, and they will be able to click on it as if it was a normal link.

You may have noticed that we're using the HTML5 `<nav></nav>` and `<main></main>` elements. Here's why:

<nav>

From **MDN**, the HTML `<nav>` element represents a section of a page whose purpose is to provide navigation links, either within the current document or to other documents. Common examples of navigation sections are menus, tables of contents, and indexes.

By placing your app navigation inside the `<nav>` element, you're also helping users using screen readers and other accessibility software to understand your app better.

From **MDN**, the HTML `<main>` element represents the dominant content of the `<body>` of a document. The main content area consists of content directly related to or expanding upon a document's central topic or the application's central functionality.

By placing your `<Routes />` and `<Route />` components inside the `<main></main>` element, you're instructing the browser, search engines, etc. that this is where the main content is on your page.

React Tutorial note

A quick note on React Tutorial: if you use `<a>` links by accident, instead of the `<Link />` component, clicking on these `<a>` links will **not** perform a page navigation. This behavior is unique to React Tutorial as we don't want you to navigate away by accident.
But know that clicking on an `<a>` in a real project will perform a page navigation.

Recap

- The `<Link />` element renders an `<a>` element (for accessibility reasons), but with some event handlers that will instruct React Router to do an instant redirect instead of a page navigation.
- It's a best practice to wrap your links with `<nav>`.
- It's a best practice to wrap your `<Routes />` & `<Route />` with `<main>`.

Was this helpful?

URL params

Last updated December 2022

What if you have a list of products, anywhere from two products all the way to a hundred (or even more). How do you create a page for every single product?

This assumes that all the products will have the same page structure, but the data will differ. For that, we use React Router's *URL parameters*.

Let's take a look at an example, and then we'll explain how it works:

```
<Route path="/products/:id" element={<Product />}>
</Route>
```

Notice how the path is `/products/:id`. The `:id` is a placeholder for the `id` that we will be able to read as a variable in the next lesson.

The colon character (`:`) instructs React Router that we should not be looking exactly for `:id` but for any text that comes after `/products/`. For example, these routes will match this path `/products/:id`:

- `/products/1` ⇒ the `:id` is **1**
- `/products/2` ⇒ the `:id` is **2**
- `/products/30` ⇒ the `:id` is **30**
- `/products/abc` ⇒ the `:id` is **abc**

All these routes will render the `<Product />` component.

Note how it doesn't necessarily have to be an `id` (a number); anything that comes after `/products/` will be caught by React Router.

This will allow you to create a page that will display the details of the product. We will see in the following lecture how to read that value.

Recap

- URL parameters allow you to show a dynamic page for a certain entity in your application. This page will receive a parameter such as an `id`.
- The syntax for describing URL parameters in React Router is the **colon**. For example, `:id` or `:name`.

Was this helpful?

UseParams hook

Last updated December 2022

In the previous lesson, we defined a route that accepts a URL parameter and renders a certain component, for example, the `<Product />` component.

But how can the `Product` component know what that URL parameter was?

Assuming the following route:

```
<Route path="/products/:id" element={<Product />}>
</Route>
```

Inside the `<Product />` component, you can extract the `:id` parameter from the URL using the `useParams()` hook.

useParams()

The `useParams` hook is a hook we need to import from "react-router-dom". This hook returns an object containing all the URL parameters from the URL:

```
// Product.js
import {useParams} from "react-router-dom";

function Product() {
  const params = useParams();
  console.log(params); // object containing params

  return <h2>Product</h2>;
}
```

You start by importing the `useParams` as a named import from `react-router-dom`. Then you call that hook at the top of the component with `const params =`

It will always return an object containing all the URL parameters from the URL. Using our current example, let's say the user visits the page: `/products/10`, then the params object will contain:

```
{
    id: "10"
}
```

This means that if we want to get the `id` of the product from the URL, we will have to access `params.id`.

URL parameter values are always strings

One important thing to note is that the values of URL parameters will always be strings. That's because they're written in the URL, and the URL is a string. So if you write a number or a boolean (such as `true` or `false`), you will get them as a *string*.

If you need to convert that id into a number, you can do so as follows:

```
// Product.js
import {useParams} from "react-router-dom";

function Product() {
  const params = useParams();
  // https://codetogo.io/how-to-convert-string-to-number-in-
javascript/
  const id = Number.parseInt(params.id, 10);

  return <h2>Product</h2>;
}
```

Multiple URL parameters

All the examples so far had only one URL parameter. However, you can have more than one URL parameter. Let's take a look at an example:

```
</Route>
```

Assuming the user visits the following URL: `/products/food/42`, the `:type` parameter will correspond to `"food"` and the `:id` parameter will correspond to `"42"`.

We can update our component to read both parameters since the `useParams()` hook returns an object of all these URL parameters:

```
// Product.js
import {useParams} from "react-router-dom";

function Product() {
  const params = useParams();
  const id = params.id; // "42"
  const type = params.type; // "food"

  return <h2>Product</h2>;
}
```

Recap

- The `useParams()` hook allows us to extract the parameters from the URL.
- You have to import the named `useParams` hook from `react-router-dom`.
- The `useParams()` hook always returns an object containing the URL parameters as key/value pairs.
- The values of the URL parameters are always in strings because they are extracted from the URL.

Was this helpful?

Project XV

Last updated December 2022

This project aims to make the product listing clickable so that every product will have its own page where the details of that product are displayed.

Here's what the final result will look like:

We will make this work by creating a `<ProductDetails />` component that reads the **id** of the product and fetches the details from an API endpoint.

There are two reasons why we need to do a fetch call inside the `<ProductDetails />` component:

1. The endpoint we're calling contains additional details such as the product price (since we're browsing the details of a specific product).
2. If the user copies the link and sends it to a friend, they will be able to view the product details even though they didn't click on the product from the homepage.

What are nested routes?

Last updated December 2022

A common pattern you see in some websites is going to the **about** page located at `/about`, which then renders a cover image, the title **About**, and then you will be able to select one of the two sub-pages:

- **About us** located at `/about/us/`
- **About the Team** located at `/about/team/`

The **About us** page would then render the *same content* as the `/about` page but with the addition of an *About us* section underneath.
Similarly, the **About the Team** page would render the *same content* as the `/about` page but with the addition of an *About the Team* section underneath.

This is an example of nested routes. That's because the `/about/us` page is rendering two routes simultaneously. It's rendering the content from the `/about` route as well as the content from the `/about/us` route.

Before we explain how you can implement it in React Router, let's take a look at a visualization challenge so that you can try these nested routes in the *browser* tab.

Recap

- Nested routes allow you to render several routes at the same time. For example, `/about/us` would render the `/about` and the `/about/us` routes simultaneously.

Was this helpful?

React Router nested routes

Last updated August 2023

Making nested routes work in React Router is a three-step process:

1. You need to start by nesting new `<Route />` definitions inside an existing `<Route />` element.
2. Then, you need to import and use an `<Outlet />` component inside the route that will have nested routes.
3. Finally, you want to make sure you have links pointing to these routes so you can view/test them.

Let's start with an example where we have an `<About />` component that renders on `/about`:

```
import {BrowserRouter, Routes, Route} from "react-router-dom";

function About() {
  return <>
        <h1>About</h1>

    </>;
}

function App() {
    return (
        <BrowserRouter>
            <Routes>
                <Route path="/about/" element={<About />}>
                </Route>
            </Routes>
        </BrowserRouter>
    );
}
```

Whenever the path starts with `/about` we want to render it as well as some child components. This will allow us to always show the `<h1>About</h1>` on sub-pages such as `/about/us` and `/about/team`.

Step 1: Define nested `<Route />` definitions

First, we'll need to define the two nested routes under `/about`. We'll have a `<Route path="us" />` and a `<Route path="team" />`. Notice that the paths are relative because we'll nest these inside the `<Route path="/about/" />`. These two `<Route />` components become children of the `<Route path="/about" />` like so:

```
<Route path="/about/" element={<About />}>
    {/* about/us */}
    <Route path="us" element={<h2>About us</h2>}>
    </Route>
    {/* about/team */}
    <Route path="team" element={<h2>About the Team</h2>}>
    </Route>
</Route>
```

We've added comments showing you that `path="us"` nested inside `path="/about/` makes the final path of that route `/about/us`. The same concept applies to `path="team"`, which becomes `/about/team`.

> Don't start the nested routes with a / character. Otherwise, you will have to repeat the `/about` path. So, `path="us"` and `path="/about/us"` are valid. However, we recommend the first approach.

2. Use an `<Outlet />`

Now, we need to instruct the `<About />` component to render those nested routes. For that, we need to import the `Outlet` component from `react-router-dom` and use

```
import {BrowserRouter, Routes, Route, Outlet} from "react-
router-dom";

function About() {
  return <>
        <h1>About</h1>

        {/* Render nested routes for /about/... here */}
        <Outlet />
    </>;
}

function App() {
    return (
        <BrowserRouter>
            <Routes>
                <Route path="/about/" element={<About />}>
                    {/* about/us */}
                    <Route path="us" element={<h2>About us</h2>}>
</Route>
                    {/* about/team */}
                    <Route path="team" element={<h2>About the
Team</h2>}></Route>
            </Route>
          </Routes>
        </BrowserRouter>
    );
}
```

Suppose you place the `<Outlet />` before the `<h1>About</h1>`. In that case, the nested routes content (`<h2>About us</h2>` or `<h2>About the Team</h2>` will render **before** the h1).

3. Add `<Link />` elements

```
import {BrowserRouter, Routes, Route, Outlet, Link} from
"react-router-dom";

function About() {
  return <>
      {/* Content always rendered when the URL starts with /about
*/}
      <h1>About</h1>
      <Link to="us">About us</Link> |
      <Link to="team">About the team</Link>

      <Outlet />
    </>;
}

function App() {
    return (
        <BrowserRouter>
          <Routes>
            <Route path="/about/" element={<About />}>
                {/* about/us */}
                <Route path="us" element={<h2>About us</h2>}>
                </Route>
                {/* about/team */}
                <Route path="team" element={<h2>About the
Team</h2>}>
                </Route>
            </Route>
            </Route>
          </Routes>
        </BrowserRouter>
    );
}
```

Let's practice what you learned in the following challenge!

Recap

- Making nested routes work in React Router is a three-step process:

1. Define nested `<Route />` definitions.
2. Use an `<Outlet />` to specify the location of nested routes on the parent component.
3. Add `<Link />` elements to view and test nested routes.

Was this helpful?

useOutletContext

Last updated July 2022

We saw in the previous lesson how to nest routes. The component that will render nested routes will have an `<Outlet />`.

There are situations where you will need to pass some data to the components rendered by the `Outlet`. We will have an example of that in the final project. For now, we'll take a basic example where we have a variable called `data` that we want to make available for any component rendered under the `<Outlet />`.

```
function About() {
    // we want to make this available to the <Outlet /> sub-
components
    const data = {
        someValue: 123
    };

    return <>
        {/* Content always rendered when the URL starts with
/about */}
        <h1>About</h1>
        <Link to="us">About us</Link> |
        <Link to="team">About the team</Link>

        <Outlet />
    </>;
}
```

For that, we can use the `context` attribute on the `<Outlet />` component, automatically creating a context for us. This is a widespread use case, which is why the React Router team has baked it into the library.

```
function About() {
    // we want to make this available to the <Outlet /> sub-
```

```
        someValue: 123
    };

    return <>
        {/* Content always rendered when the URL starts with
/about */}
        <h1>About</h1>
        <Link to="us">About us</Link> |
        <Link to="team">About the team</Link>

        // pass it to Outlet as a context
        <Outlet context={data} />
    </>;
}
```

This `context` attribute is special to React Router and is only available on the `<Outlet />` component. Now React Router has automatically created a context for us that is available for the children of `<Outlet />`. We just have to use it.

useOutletContext hook

This is where the `useOutletContext` hook comes into place. We can use it in any component expected to render under the `<Outlet />`. For example, the `AboutUs` component:

```
import {useOutletContext} from "react-router-dom";

function AboutUs() {
    const context = useOutletContext();
    console.log(context); // {someValue: 123}
    console.log(context.someValue); // 123

    // ...
}
```

Recap

- It's common to need to pass data to an `<Outlet>`. This data will be accessible by the components rendered under the `Outlet`. React Router automatically creates a Context for you when you use the `context` attribute on the `<Outlet />` component.
- To read the data from the Outlet's context, you must use the `useOutletContext` hook.

Was this helpful?

Handling 404

Last updated December 2022

For larger projects, handling routes that are not found becomes important. These are sometimes called 404 routes because of the **HTTP status code** 404 when a document is not found on the server. (If you like cats, you'd enjoy **http.cat**)

Let's see how we can create a Route that will match whenever no other route matches the current browser URL:

```
import {BrowserRouter, Routes, Route} from "react-router-dom";

function App() {
    return <BrowserRouter>
        <Routes>
            <Route path="/" element={<p>Landing page here</p>}>
</Route>
            <Route path="/products" element={<p>Products page
here</p>}></Route>
            <Route path="*" element={<p>404 page here</p>}>
</Route>
        </Routes>
    </BrowserRouter>
}
```

Now when the user goes to the `/`, they will see `Landing page here`. Similarly, when they go to `/products` they will see `Products page here`. And for all other links, they will see `404 page here`.

Why is that? That's because the `<Route>` that we created for the 404 page has the `path` prop set to `*`. React Router will match this route only when no other route can be found.

The order of the 404 Route does not matter. You can place it anywhere within the `<Routes> ... </Routes>` declaration as a direct descendant.

Recap

- A route with `path="*"` will only match when no other route has been found. This is often used to create a 404 page.

Was this helpful?

Active page

Last updated December 2023

Another everyday use case when an app contains a route is highlighting the currently active page in the menu.
Say you have a navigation menu that has two links:

1. **Home** that takes the user to `/`
2. **About** that takes the user to `/about`

Then it is possible to automatically highlight the **About** link with React Router when the user is on the `/about` route. Similarly, you can highlight the **Home** when the user is on the `/` route. All of that without having to write conditionals and complicated logic.
Here's how:

```
import {NavLink} from "react-router-dom";

function getClassName({isActive}) {
    if (isActive) {
        return "active"; // CSS class
    }
}

function App() {
    return <ul>
        <li>
            <NavLink to="/" className=
{getClassName}>Home</NavLink>
        </li>
        <li>
            <NavLink to="/about" className=
{getClassName}>About</NavLink>
        </li>
    </ul>
}
```

active page's menu link highlighted in bold.

```
.active {
    font-weight: bold;
}
```

The `className` prop in the `NavLink` component can also accept a function (so, not just a string). This function is automatically called by React Router with an argument, which is an object containing various properties about the navigation state. One of these properties is `isActive`. We're immediately destructuring that in the function parameter, which is why we have `{isActive}` in the parameter.

Then, we check if `isActive` is truthy. In that case, we return the name of the CSS class, `active`.

The `isActive` returns a boolean when the current React Router path is the same as the `NavLink`'s `to` attribute.

React Router will now automatically add the class `active` to the `<NavLink />` corresponding to the currently active route.

Shorter version

If you use the ternary operator, there's a shorter version to write the above. If you've taken the [Learn JavaScript]() course, you may have noticed that we completely skipped the ternary operator. That's because it often negatively impacts readability. However, the following use case is considered acceptable.

The ternary operator allows you to replace an `if/else` statement using the `condition ? truthy expression : falsy expression`.

The following JavaScript:

```
function getClassName({isActive}) {
    if (isActive) {
        return "active";
```

```
    }
}
```

can be replaced with:

```
function getClassName({isActive}) {
    return isActive ? "active" : "";
}
```

You just have to make sure to put the `return` outside of the ternary. When `isActive` is truthy, the expression after `?` will execute (`"active"`). Otherwise, the expression after the `:` will execute (`""`).

Knowing that, we can completely remove the separate function definition `getClassName` and write the `isActive ? "active" : ""` inside an arrow function `({isActive}) => isActive ? "active" : ""`.

If you find this too much of a jump, feel free to continue defining a separate function.

The final result looks like the below:

```
import {NavLink} from "react-router-dom";

function App() {
    return <ul>
        <li>
            <NavLink to="/" className={({isActive}) => isActive ?
"active" : ""}>Home</NavLink>
        </li>
        <li>
            <NavLink to="/about" className={({isActive}) =>
isActive ? "active" : ""}>About</NavLink>
        </li>
    </ul>
}
```

You may have noticed that we're using `NavLink` rather than `Link`.

This is **very** important as the `<Link />` component does **not** accept a function definition for the `className` prop.

A common mistake is to use the `ClassName` with a function definition on the `<Link />` component, but that won't work!

Recap

- The `className` prop, when used on a `NavLink`, accepts a function definition that lets you set the class based on `isActive` that is true when the current route matches the NavLink's `to` prop.

Was this helpful?

useNavigate

Last updated December 2022

You may need to move pages within your React application programmatically in some scenarios.
For example, you send a `fetch` request, and based on that result, you'd like to redirect the user to a certain page.

Here are some more common examples:

- Redirect the user to the `/login` page if they're not logged in.
- Redirect the user to the `/dashboard` after the `fetch()` request for login returns successfully.

For that, you'd need to use the `useNavigate()` hook, which returns a method you can call to navigate to a new route programmatically. Let's take a look:

```
import React, {useEffect} from "react";
import {useNavigate} from "react-router-dom";

function HelpPage() {
    const navigate = useNavigate();

    useEffect(() => {
        const isLoggedIn =
window.localStorage.getItem("isLoggedIn");
        if (!isLoggedIn) {
            navigate("/login");
        }
    }, []);

    return <h2>Help page</h2>;
}
```

calling `navigate("/login")`.

It only works inside BrowserRouter

You need to make sure to call the `useNavigate()` hook in a component that is nested **inside** `<BrowserRouter />`, or else it won't work.

The example below will NOT work:

```
import React from "react";
import {BrowserRouter, useNavigate} from "react-router-dom";

function App() {
    // ❌ This breaks because the component was not wrapped by
BrowserRouter, but its children are.
    const history = useNavigate();

    return <BrowserRouter>
        {/* ... */}
    </BrowserRouter>;
}
```

As you can see, the `useNavigate()` hook won't work inside the `App()` component because it was not wrapped by `<BrowserRouter />`. However, all the child components inside App will be wrapped by `<BrowserRouter />`, so you can use `useNavigate()` in those child components.

If you need to use `useNavigate()` in the `<App />` component, then you need to create a parent component that wraps `<App />` and move the `<BrowserRouter />` to that parent component.

Accessibility

All your links should be `<Link />` elements because they will render as an `<a>` tag with proper accessibility.

The `push()` method should only be used whenever a `<Link />` is not possible. That means whenever the redirect is **programmatic** (that means it's based on some logic, such as the result of `fetch()`).

Recap

- Programmatic redirects are redirects based on some sort of checks, such as based on the result of `fetch()` or a conditional.
- You can programmatically change the current route with React Router's `useNavigate()` hook, which you can call with the path you want to navigate to.
- For accessibility reasons, try to use `<Link />` whenever possible.

Was this helpful?

useLocation

Last updated December 2022

What if you need to run a piece of code whenever React Router navigates to a new URL?

You can do that with the `useLocation` hook.

But first, let's explore some use cases. The most common use case for this is to send a **pageview** event to your analytics service, such as Google Analytics. Note that as of Google Analytics 4, you don't need to do that, as it will be done automatically by Google Analytics itself. But we'd still like to teach you the `useLocation` hook as you may need to use it for other purposes.

Such as, for example, integrating with other libraries and analytics services, running conditional logic for some routes, etc.

useLocation hook

The `useLocation` hook returns information about the **currently** active route.

Here's how you use the `useLocation` hook:

```
import {BrowserRouter, Routes, Route, useLocation} from "react-
router-dom";
import {createRoot} from "react-dom/client";

function App() {
    const location = useLocation();
    console.log(location.pathname);

    return <Routes>
        {/* routes here ... */}
    </Routes>
}
```

```
root")).render(<BrowserRouter><App />></BrowserRouter>);
```

The `location` variable will be an object containing the `pathname`.

You can then use the `location.pathname` to know the current route the user is browsing. For example, `/` or `/about`, etc., depending on your routes.

Just like the `useNavigate` hook, you need to make sure that the code is running **inside** a component wrapped by `<BrowserRouter />` or else it won't work.

On location change

Knowing how `useLocation` works and how to get the current `pathname`, you can run a piece of code whenever React Router navigates to a new URL by wrapping your code with a `useEffect` with a dependency on `location`. Here's how:

```
import React, {useEffect} from "react";
import {BrowserRouter, Routes, Route, useLocation} from "react-
router-dom";
import {createRoot} from "react-dom/client";

function App() {
    const location = useLocation();

    // run a piece of code on location change
    useEffect(() => {
        console.log(location.pathname);
        // send it to analytic, or do some conditional logic here
    }, [location]);

    return <Routes>
        {/* routes here ... */}
    </Routes>
}
```

The code inside the `useEffect` hook will run whenever the `location` object changes, which happens whenever the user navigates to a new route.

Recap

- The `useLocation()` hook returns information about the currently active route.
- The `useLocation()` hook returns the `pathname` of the current route.
- By combining `useLocation()` with `useEffect()`, you can run a piece of code whenever the user navigates to a new route.

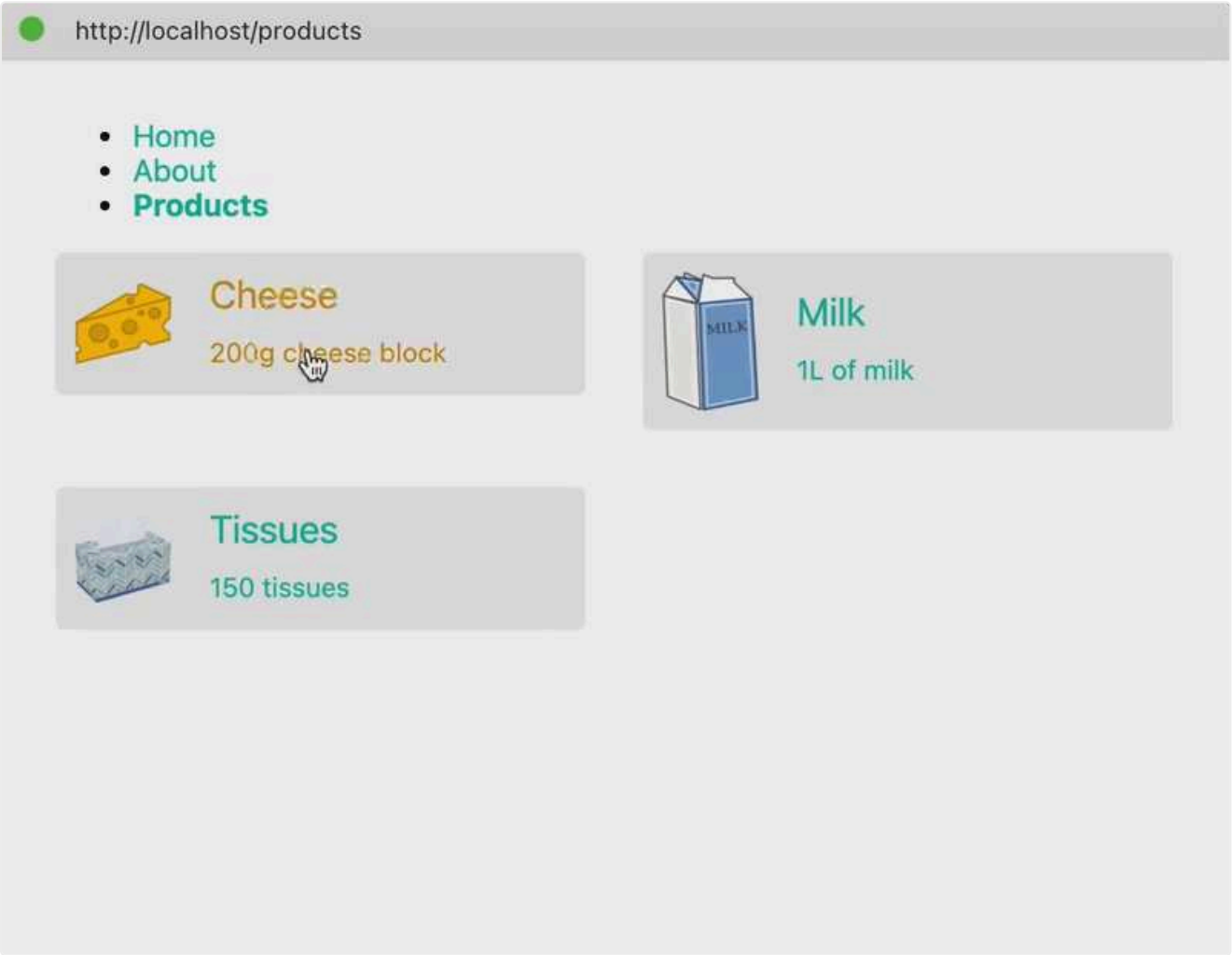
Was this helpful?

Online store setup

Last updated December 2022

This project will use some of the most important concepts you learned in the last two React Router chapters so that you're ready to work on the final project!

For now, let's iron out those React Router concepts!



Was this helpful?

Project recap

Last updated July 2022

Great job!

Back in the **first React Router lesson**, we said the following:

> The `react-router-dom` package exports many components and hooks. However, there are 10 of those that you will often use. Throughout the following few chapters, we will teach you these 10 components and hooks in this course. We will take the same step-by-step approach that you're used to by now.

As a recap, here's what you learned, split into two categories:

Most common

1. `<BrowserRouter />`
2. `<Routes />`
3. `<Route />`
4. `<Link />`
5. `useParams()`
6. `<Outlet />`

Secondary

7. `<NavLink />`
8. `useNavigate()`
9. `useLocation()`
10. `useOutletContext()`

page. See you there!

Was this helpful?

Introduction to Stripe

Last updated March 2023

Stripe is a payment processor that allows you to process payments (for example, credit/debit cards) from your users. It is available in the **following countries**.

There are alternatives to Stripe, such as **Braintree**, which compares well to Stripe. Alternatively, you can accept PayPal payments with **Paypal**.

In this course, you will **not** be required to create a Stripe account.

Accepting payments

There are several ways to accept payments with Stripe. This depends on the kind of payment you accept and how you plan to handle it. For example, is it a one-time payment or a recurrent one? Should the user immediately get access to the product/service they paid for? Or can you process it manually in the Stripe dashboard?

In our case, we accept a **one-time payment** from the user when they purchase the products from the supermarket. Furthermore, we will be manually processing their order via the Stripe Dashboard. That means we will see the new order in the Stripe Dashboard and manually initiate the delivery.

For that, we can use **Stripe Checkout**. You can try it out here: **checkout.stripe.dev**.

Stripe Checkout works with the least amount of configuration. Because we are processing the orders manually via the Stripe dashboard, we won't need to have any server integration.

Live vs. Test modes

When integrating Stripe, you can use the **test** mode to test your payments without actually withdrawing money from credit cards (you won't even need an actual credit

You can switch to the **live** mode when you're ready. Once you are in live mode, then real money will be withdrawn from credit cards.

When you're in **test** mode, you can use the following credit card details:

- Card number: `4242 4242 4242 4242`
- Month: Any valid month (1 to 12)
- Year: Any valid year (current year or above)
- CVC: Any 3 digits (for example, 123)

Throughout this course, we will use the **test** mode, which means you can use the above card number to test your payment. The card number will be given again to you in the challenge.

So how do you tell Stripe that you want to use the **test** mode rather than the **live** mode? It all comes down to the **API key** that you pass to Stripe.
There are different API keys for each mode.

Throughout this course, we will give you a **test** API key that you can use **only** in this course. You are not allowed to use it outside of this course, and there is no guarantee that it will keep working afterward, as we will be re-shuffling it often.

Recap

- Stripe is a payment processor that allows you to process payments (for example, credit/debit cards) from your users.
- **Stripe Checkout** works with the least amount of configuration, and because we are processing the orders manually via the Stripe dashboard, we won't need to have any server integration.
- Stripe has a **test** mode that allows you to test your integration before going **live**.
- When in **test** mode, you can pay using the test credit card with the number `4242 4242 4242 4242`.

Creating the products

Last updated July 2022

When using Stripe Checkout, you will need to create the `products` that you want to sell in the Stripe dashboard. We have already created a product that you will use in this project.
If you'd like to learn how to do that yourself, then follow the process below:

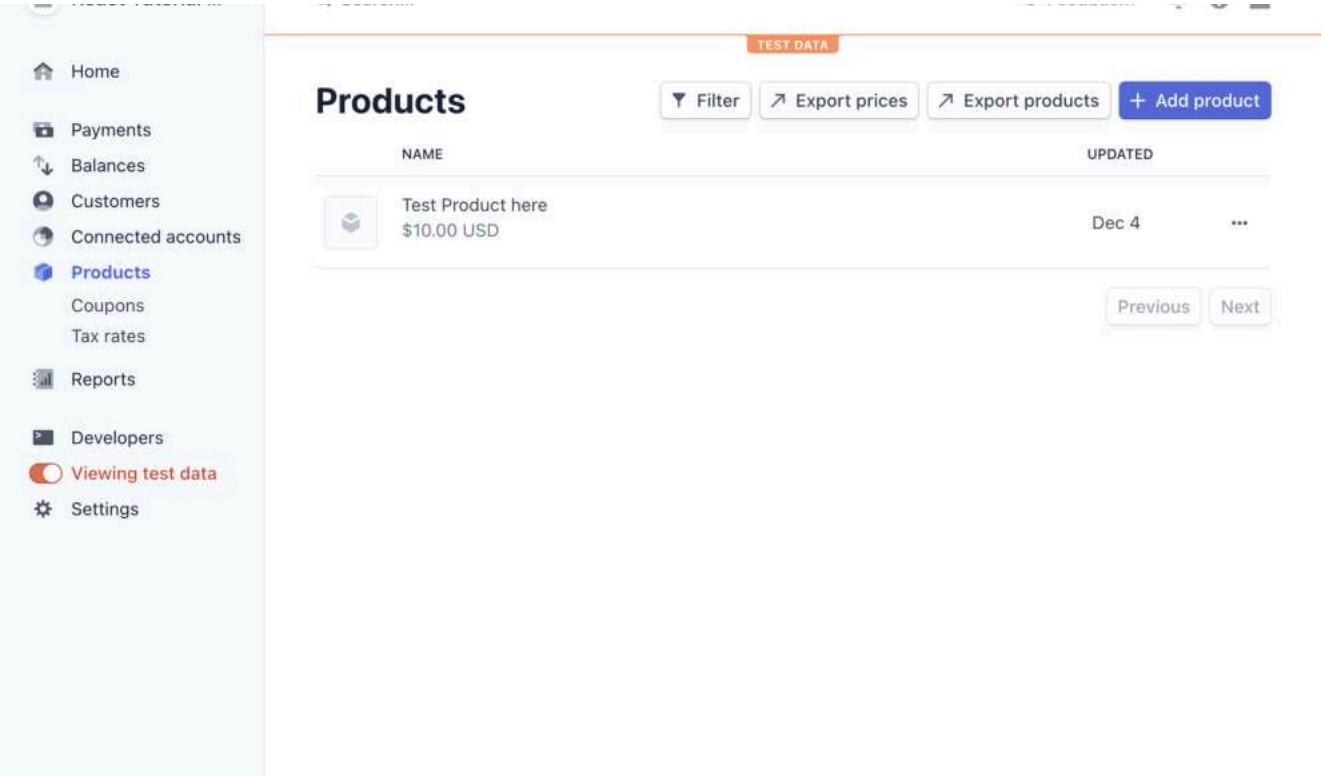
Step 0: One-time setup

This step needs to be done once. You have to **verify** your account with Stripe by providing your business details. Then, you have to go to the Checkout settings by following this link: [dashboard.stripe.com/settings/checkout](), where you need to scroll down and click on the **Enable client-only integration**.

This will allow you to use Stripe Checkout without any back-end. Remember that this is not always possible/recommended as it depends on your use case and how you plan on processing the payment.

Step 1: Go to (Test) products

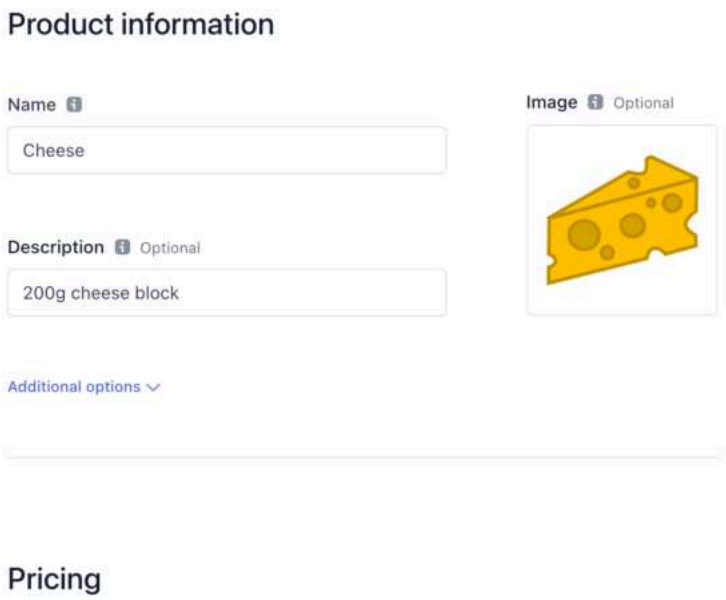
Navigate to [dashboard.stripe.com/test/products](), which takes you to the products page in the **test** environment.



On this page, you need to click on the **Add product** button.

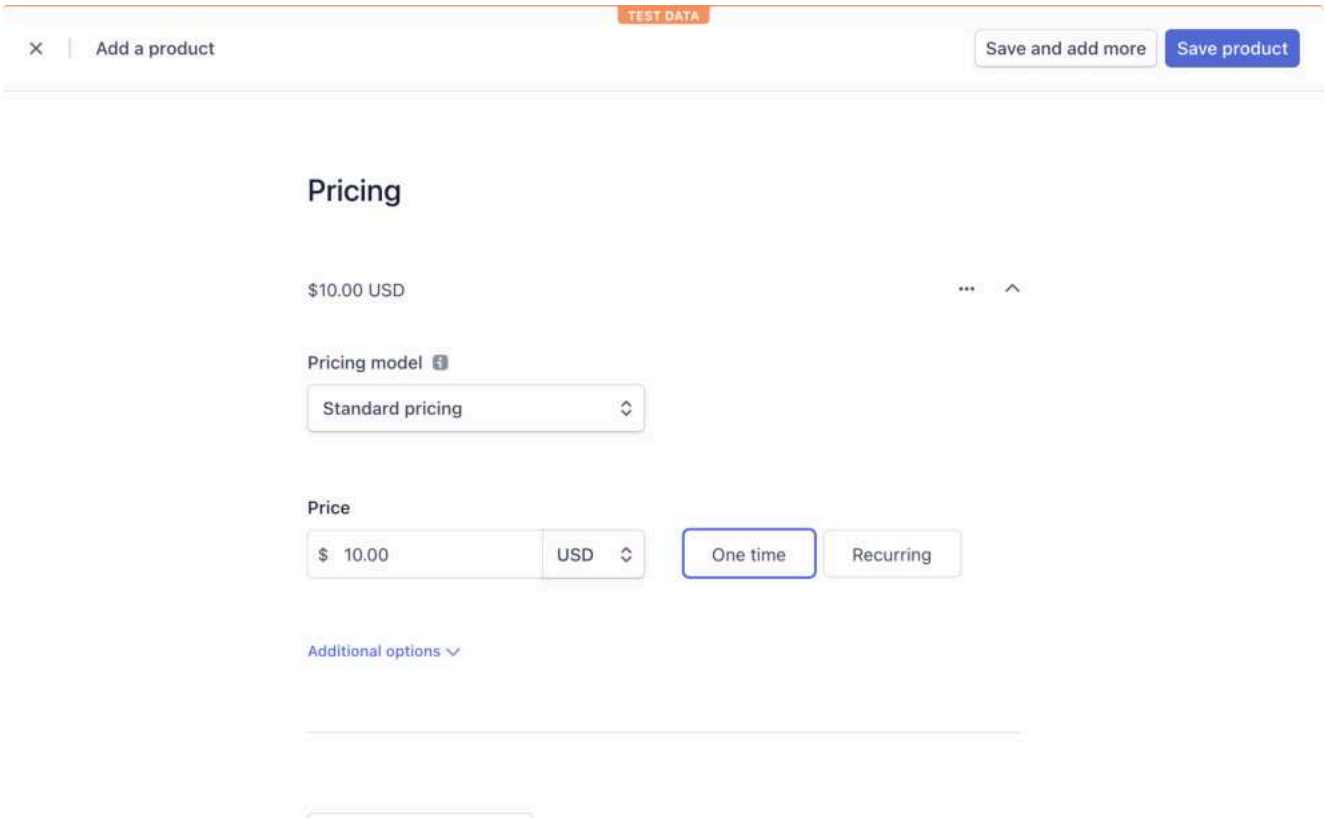
Step 2: Add product

On this page, you can describe the product by entering the name, description, and image.



Afterward, you need to scroll down and describe the **pricing** for this product. Is it a one-time payment or a recurrent one?

We've chosen a one-time payment of USD10:

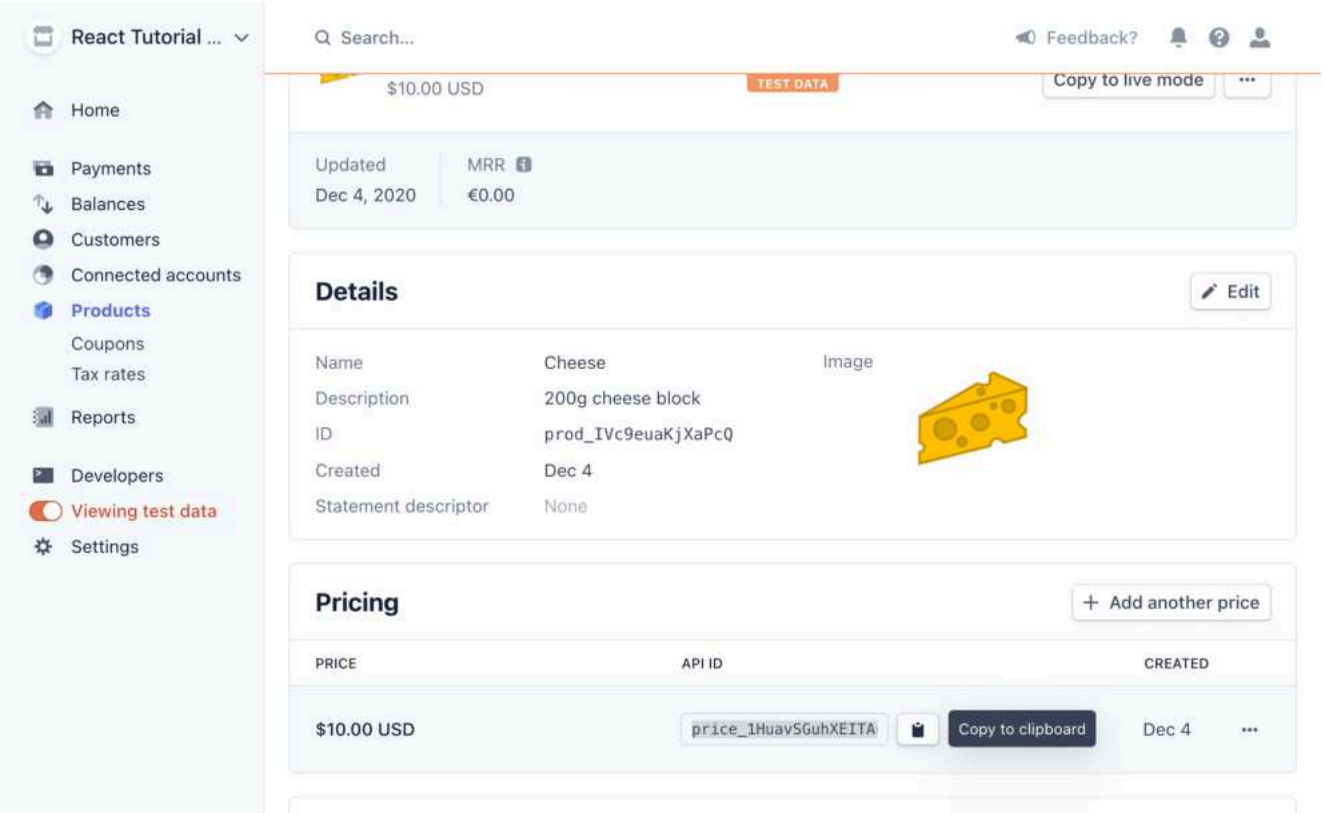


When you're done, click on the **Save product** button.

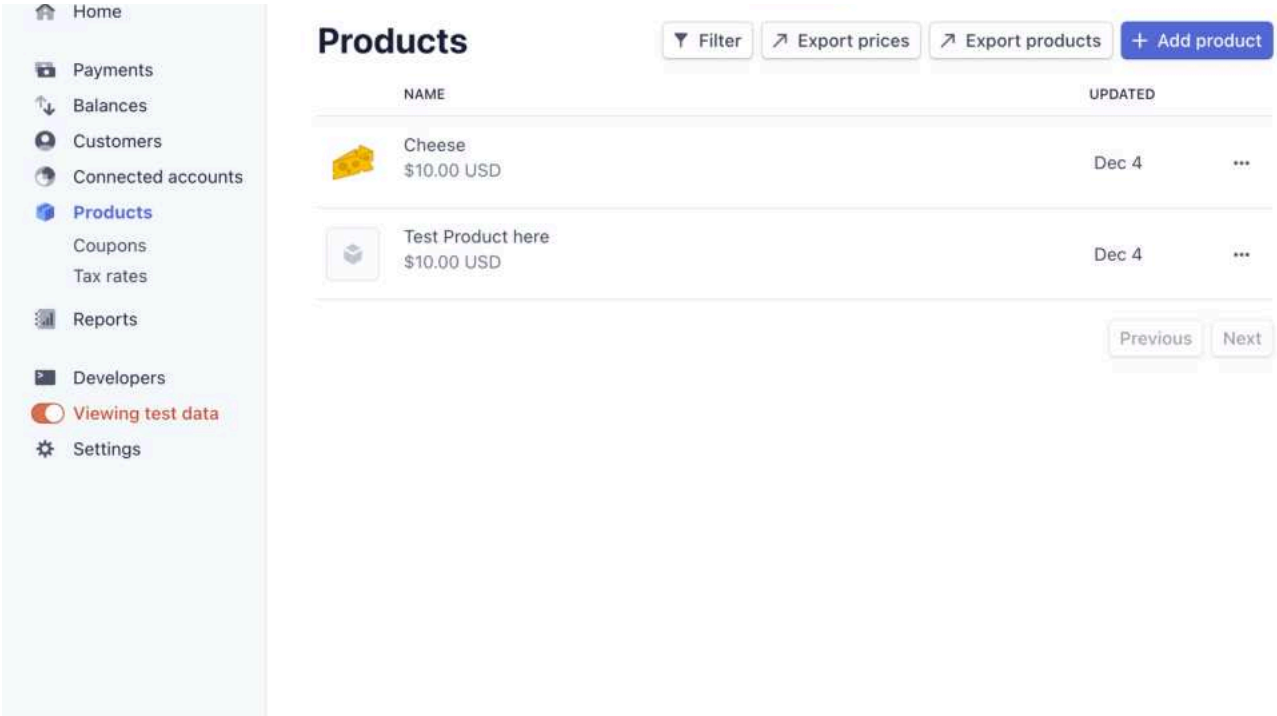
After saving the product, you will land on the product preview page, where you can see all the details you entered.

When you scroll down a little bit, you will find a section on **Pricing**. Over there, you need to copy the **price_id** as we do in the screenshot. This `price_id` will communicate with Stripe Checkout which product (and pricing plan) the user is purchasing.

We will provide you with our own `price_id`, but if you'd like to follow with your own, then make sure to copy this one.



Once you return to the **Products** page, you will see your product(s) listed on this page.



You can add multiple products. We will work with Stripe Checkout for multiple products in the final project.

Recap

- The `price_id` will be used to communicate with Stripe Checkout which product (and pricing plan) the user is purchasing.

Was this helpful?

Integrating stripe-js

Last updated July 2022

Now that we've created the product in the Stripe Dashboard, it's time to integrate stripe-js (the stripe JavaScript library) into our app.

stripe-js is the official JavaScript stripe library that loads the Stripe scripts into our app. There is also a React-specific library, **react-stripe-js**; however, since our use case is basic, we don't really need that library. Also, note that you still have to load stripe-js when using react-stripe-js. Thus, it's a good experience to learn how to use stripe-js. And that's exactly what we will be working on within this course.

importing stripe-js

The `stripe-js` package exports a named function called `loadStripe`. Here's how you import it:

```
import {loadStripe} from "@stripe/stripe-js";
```

This `loadStripe` function takes your **Stripe Publishable key** as an argument, which you can find in your Stripe > Switch to Test mode > Developers > **API keys page**. Don't forget to switch to **Test** mode so that you can test the payments with the testing credit card.

This `loadStripe` function returns a promise fulfilled when the Stripe library has loaded. Start by calling the `loadStripe` function outside your component and store its result (a promise) in a variable. We'll use that variable later on when the user clicks on the **pay** button.

```
import {loadStripe} from "@stripe/stripe-js";

const stripeLoadedPromise =
```

```
function App() {
    return <button>Pay</button>;
}
```

Don't forget to replace the `PK_REPLACE_WITH_STRIPE_PUBLISHABLE_KEY` with your own publishable key. For the challenges on React Tutorial, we will give you a test publishable key that you can use.

Next, we need to redirect the user to the hosted **Stripe Checkout** page when they click the payment button. We'll show you how to do that with promises and `async/await`. Remember that async/await is just syntactic sugar on top of promises. Feel free to choose whichever approach you prefer.

Approach #1: Promises

```
import {loadStripe} from "@stripe/stripe-js";

const stripeLoadedPromise =
loadStripe('PK_REPLACE_WITH_STRIPE_PUBLISHABLE_KEY');

function App() {
  function handleClick(event) {
    stripeLoadedPromise.then(stripe => {
      stripe.redirectToCheckout({
        lineItems: [{
          price: 'REPLACE_WITH_PRICE_ID_HERE',
          quantity: 1,
        }],
        mode: 'payment',
        successUrl: 'https://react-tutorial.app/app.html',
        cancelUrl: 'https://react-tutorial.app/app.html',
      }).then(response => {
        // this will only log if the redirect did not work
        console.log(response.error);
      }).catch(error => {
        // wrong API key? you will see the error message here
        console.log(error);
```

```
  };

  return <button onClick={handleClick}>Pay</button>;
}
```

Approach #2: async/await

```
import {loadStripe} from "@stripe/stripe-js";

const stripeLoadedPromise =
loadStripe('PK_REPLACE_WITH_STRIPE_PUBLISHABLE_KEY');

function App() {
  async function handleClick (event) {
    const stripe = await stripeLoadedPromise;
    try {
        const result = await stripe.redirectToCheckout({
          lineItems: [{
            price: 'PRICE_ID_HERE',
            quantity: 1,
          }],
          mode: 'payment',
          successUrl: 'https://react-tutorial.app/app.html',
          cancelUrl: 'https://react-tutorial.app/app.html',
        });
        // this will only log if the redirect did not work
        console.log(result.error);
    } catch (error) {
        // wrong API key? you will see the error message here
        console.log(error);
    }
  };

  return <button onClick={handleClick}>Pay</button>;
}
```

In both approaches above, we're waiting for the `stripeLoadedPromise` to fulfill successfully before calling the `stripe.redirectToCheckout` method.

This method takes several parameters; the most important is the `lineItems` which is an **array** of items. Even if you'd like to sell just one item, make sure it's an array of objects representing the `price` and the `quantity`.

The `price` will refer to the **price_id** that we created in the previous step. So make sure to replace `PRICE_ID_HERE` with your own **price_id**. On React Tutorial, you will be given a test `price_id`.

successUrl and cancelUrl

The `successUrl` and `cancelUrl` you provide will be used by Stripe to redirect the user from the hosted Stripe Checkout page back to your website. You can provide different URLs for success and error scenarios. However, be careful not to rely on the URL to validate the payment.
The user can simply browse to your `successUrl` without paying. So landing on the `successUrl` does not necessarily mean that the user paid successfully.

Usually, this is handled by the back-end that you'd have. The `successUrl` will contain payment information which you can then query using the Stripe API to double-check that the user has indeed paid.

In our scenario, we're fulfilling orders from the Stripe Dashboard so we don't need to worry about this. This is why we provide the `https://react-tutorial.app/app.html` URL so that we're back into the challenge that we were working on after the payment has been completed.

Recap

- stripe-js is the official JavaScript stripe library that loads the Stripe scripts into our app.

Final Project: intro

Last updated December 2022

Congratulations on making it to the final project!

This will be a **multi-step** project where you build the final supermarket app. Here's how it looks like (click to play, 800kb):

0:00 / 0:51

The final project will be split into 6 main categories. You will have a short lesson explaining what you will have to do in the next project(s).

The projects will be combined together. So most projects will use the solution from the previous project so that in the end, you have the final supermarket app built ✨.

Since we're building a full app, you may find it helpful to **maximize** the browser preview tab. You can do that by clicking on the green circle in the top-left corner of the browser preview tab.

Was this helpful?

UI Kit

Last updated July 2022

We will start our project by creating two components that will represent our **UI Kit**. These components are `<Button />` and `<Input />`.

Button

The goal of the `<Button />` component is to render a `button` element with several variations. The exact requirements will be given in the project.

The Button component will have two variations: Normal and Outline. Here's how we expect to use those buttons as well as how they look like:

```
<Button>Normal</Button>
<Button outline>Outline</Button>
```

The Button component should also keep in mind that we may need to specify other props, such as our own `className` or a `disabled` state. Here's an example of a valid way to use the Button component:

```
let isLoading = false;

function handleCheckoutClick(event) {
    //
}

<Button outline className="checkout-btn" disabled={isLoading}
onClick={handleCheckoutClick}>Check out</Button>
```

The `<Input />` component will be used to render an accessible text box element. So the `<Input />` component will render a `<label />` as well as an `<input />` element.

For example, `<Input placeholder="First name" />` will render the following HTML:

```
<label class="label">
    First Name
    <div>
        <input type="text" placeholder="First Name"
class="input">
    </div>
</label>
```

Notice how `<Input />` only needs to know the `placeholder` prop and will then render a `<label />` that wraps the `<input />`. By wrapping the `input` with a `label`, the form input becomes accessible without needing to provide an `id` and a `htmlFor` prop.

Furthermore, notice how the input's `type` prop defaults to `"text"` and sets the `class` to `"input"`.

A minor variation also exists; if you pass the `required` prop, the input becomes required, as seen in the screenshot above.

The exact details will be given in the project.

This UI Kit will make it easier for us to use buttons and inputs while ensuring they adhere to the same style guide without sacrificing customizability.

Was this helpful?

Basic routing & static pages

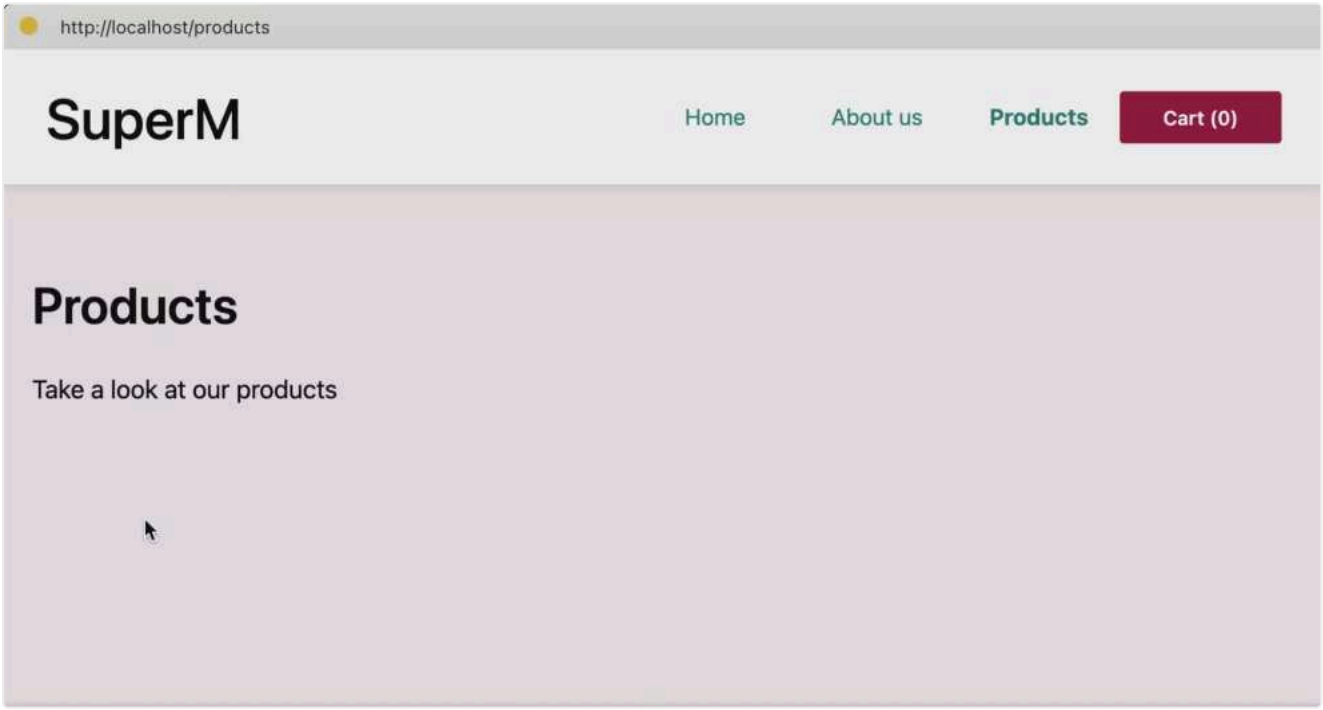
Last updated July 2022

This section aims to set up the basic routing of our website. We'll create the following pages:

- Home
- About
- Products
- Cart

The **Home** and **About** pages are static pages. Whereas the **Products** and **Carts** pages are dynamic. So we'll start by writing a placeholder on these two pages. We will provide you with the HTML for all these pages and the styles.

Knowing there are multiple pages, you will also have to set up the app's routing and navigation in the `<Navbar />` component.

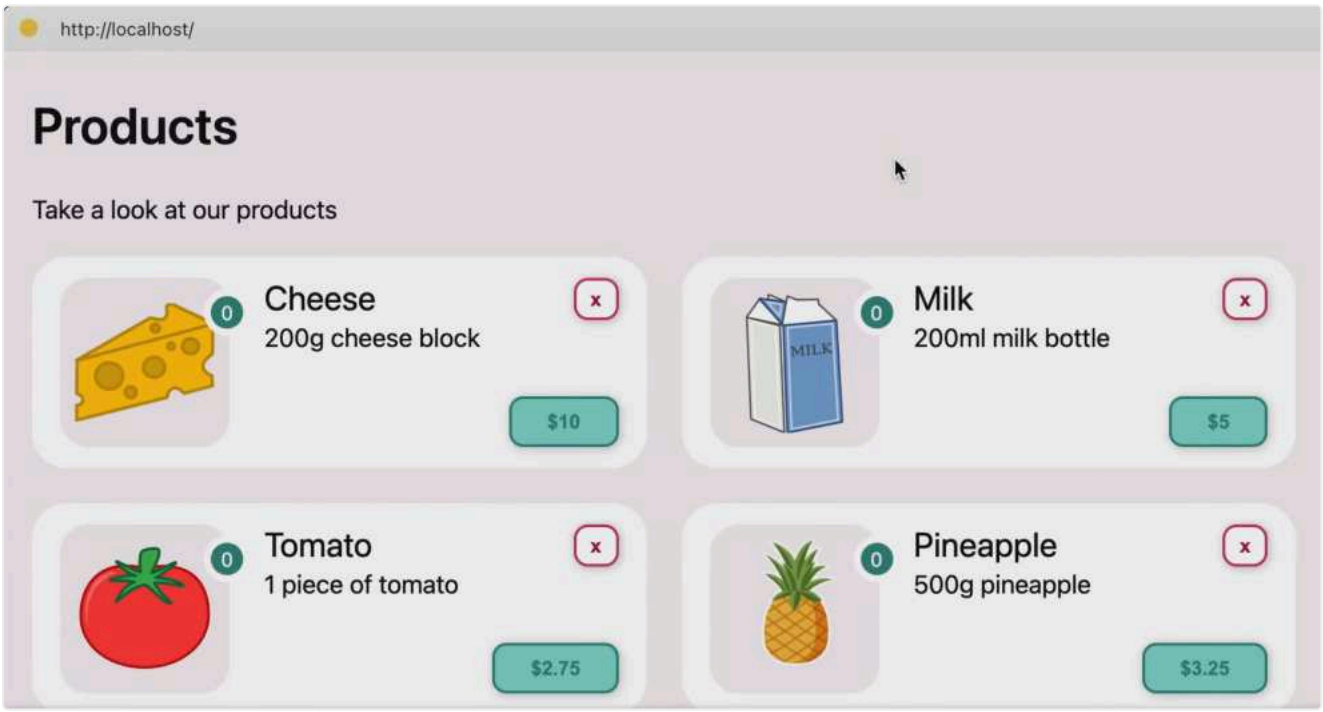


Fetching the products

Last updated July 2022

Now that we've set up the basic navigation, it's time to fetch the products from the API. There will be 2 API endpoints in the final project — one for listing the products and another for viewing a particular product by id.

We have them separately because as the number of products grows, the API response will grow. So we eliminated the fields we don't show on the product listing page, such as the nutrition details and the storage information.



So the product listing endpoint provides the list of products with details such as the name, description, image, and other properties. And the product details endpoint provides the full details of a particular product that you can query by id.

The Base URL and endpoint for listing the products will be given in the project. Don't forget to use the `useFetch` custom hook to make your life easier.

Stripe's price_id

page to Stripe Checkout.

This means that we already went ahead and created a new product on Stripe for every product you get back from the API.

Each product will have a different `price_id`. This will allow the customer to pay for the specific products they ordered once we complete the Stripe integration.

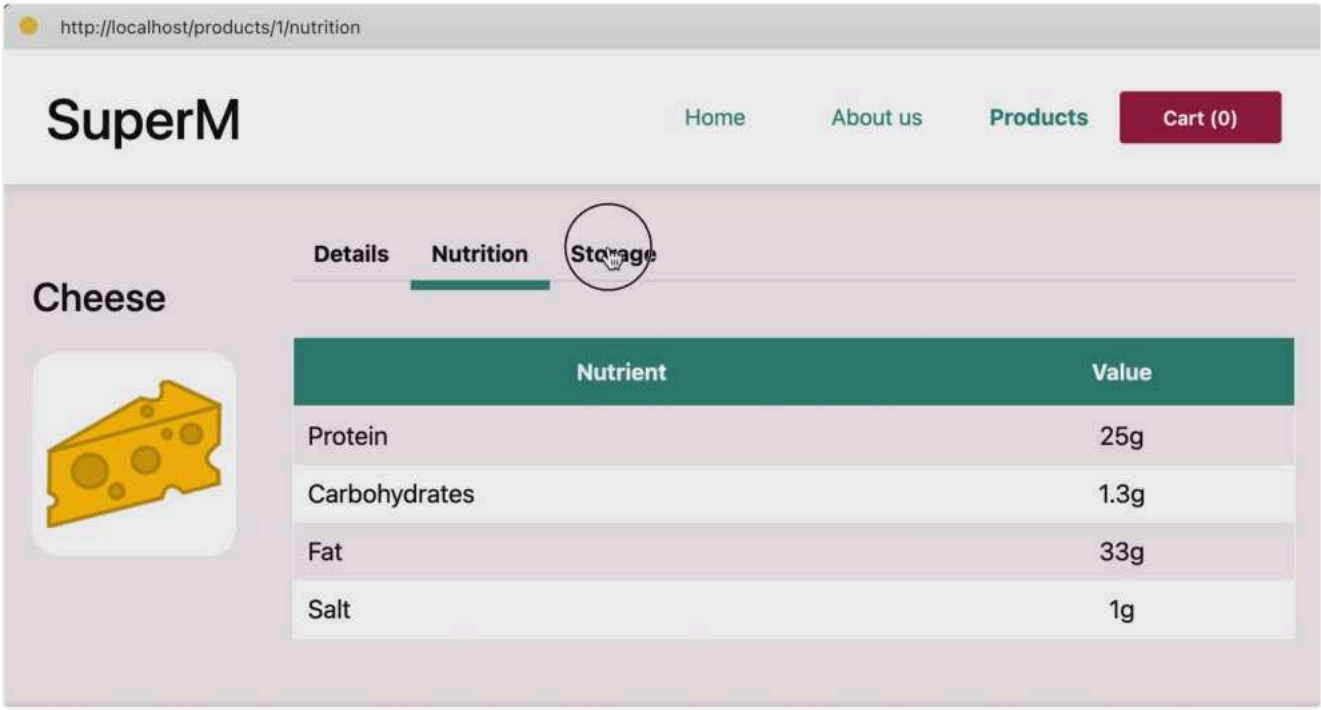
Was this helpful?

Showing the product details

Last updated December 2022

In the following project, we will allow the user to click on the product's image to view the details of the product.
This will take the user to another page. For example, clicking on the product with `id=1` will take the user to `/products/1`.

On that page, you will see a bigger version of the image and more details, such as the nutrition chart and storage information.
To get the additional details of the product, such as the nutrition chart and storage information of a specific product, you need to fetch these data from another endpoint. The endpoint will be provided in the next project.



As you can see, each tab corresponds to a route, allowing the client to share or bookmark a link to that specific view.
For example, the product's nutrition with `id=2` is located at `/products/2/nutrition`.

Sub-tabs edge case

shown by default. This means its path will be set to an empty string.

You will encounter a small issue with this when you implement the `NavLink`'s `className`. The first link will be considered `isActive` whenever any other tab is open too. That's because the `path` has been left empty.

Luckily, there's a fix to this behavior. By using the `end` prop on the `NavLink`, it will make sure not to mark this component as `isActive` when other nested paths are matched. So, when the *nutrition* tab is matched, the *details* won't be matched anymore.

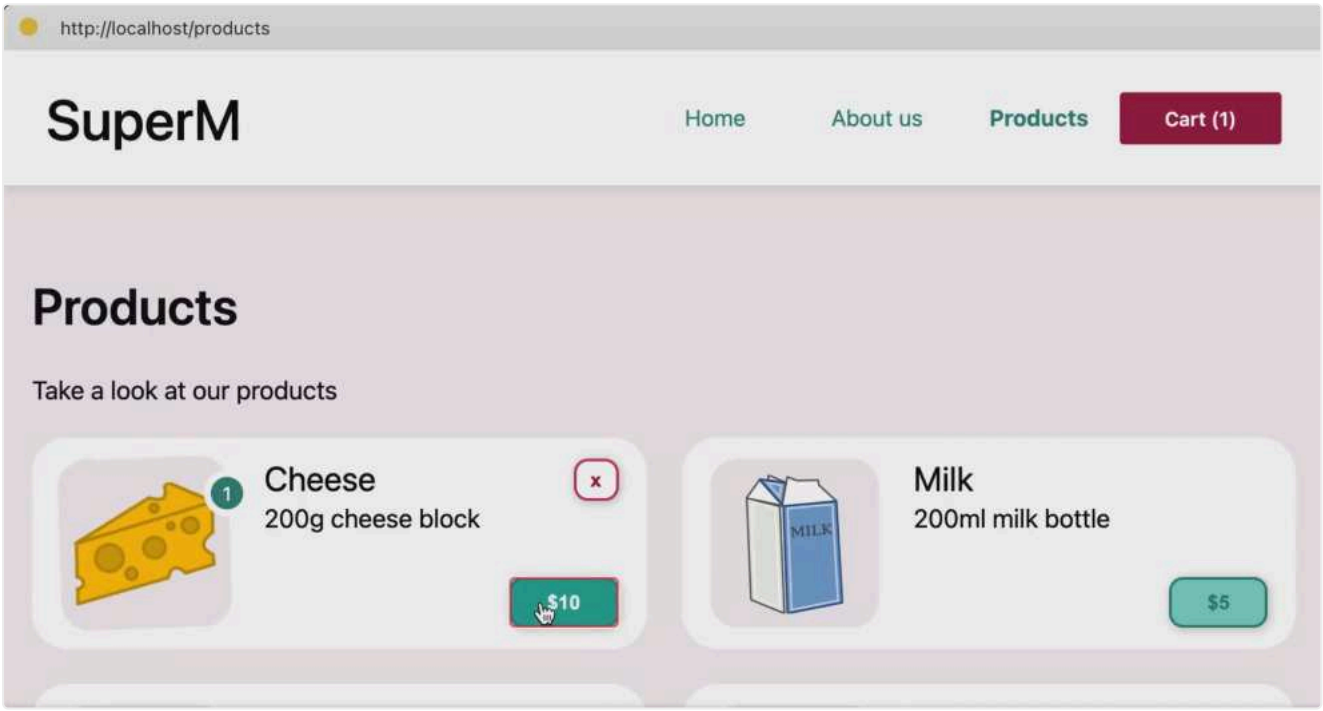
Was this helpful?

Managing cart state

Last updated July 2022

In order to build the **cart** functionality into the project, we'll need to create a state that represents the `cart`.

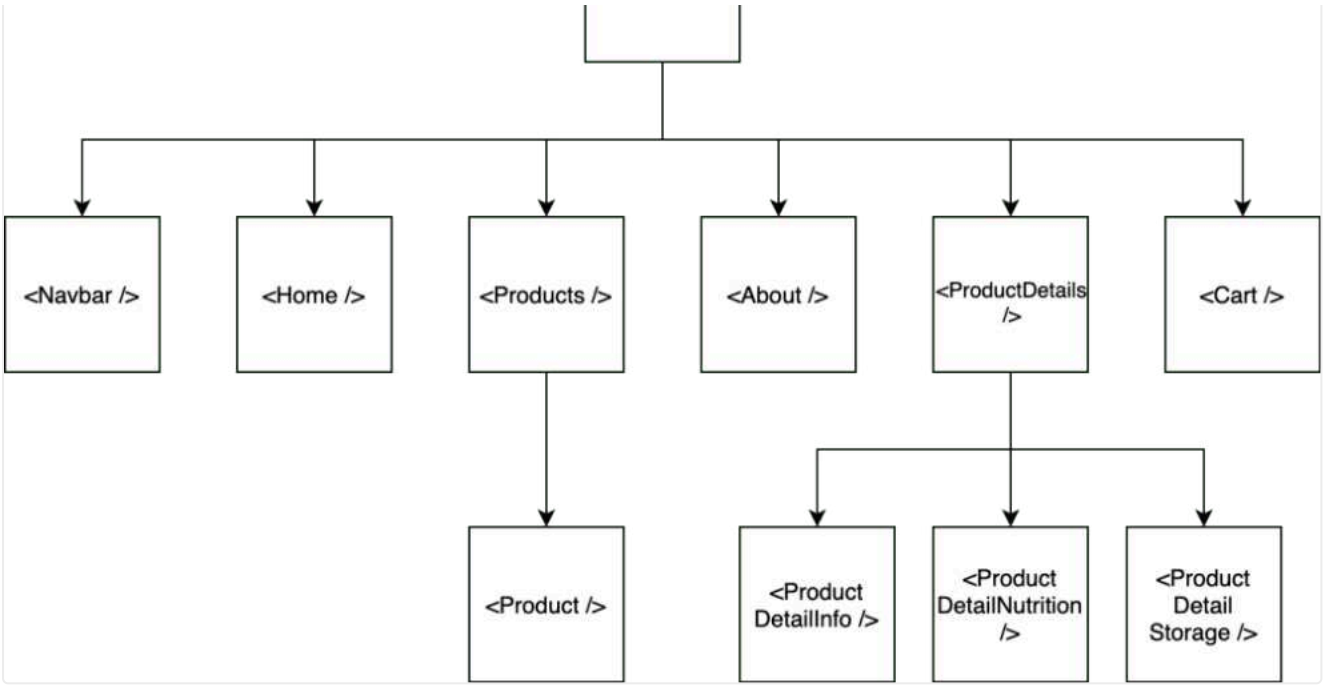
This will allow us to add items to the cart, remove items from the cart, and finally, checkout the products in the cart using Stripe.



Looking at the screencast above, the `cart` is needed in the following components:

- `<Product />` to be able to add to cart and remove from the cart
- `<Cart />` to be able to display the items in the cart and proceed to pay with Stripe
- `<Navbar />` to be able to show the number of items in the cart

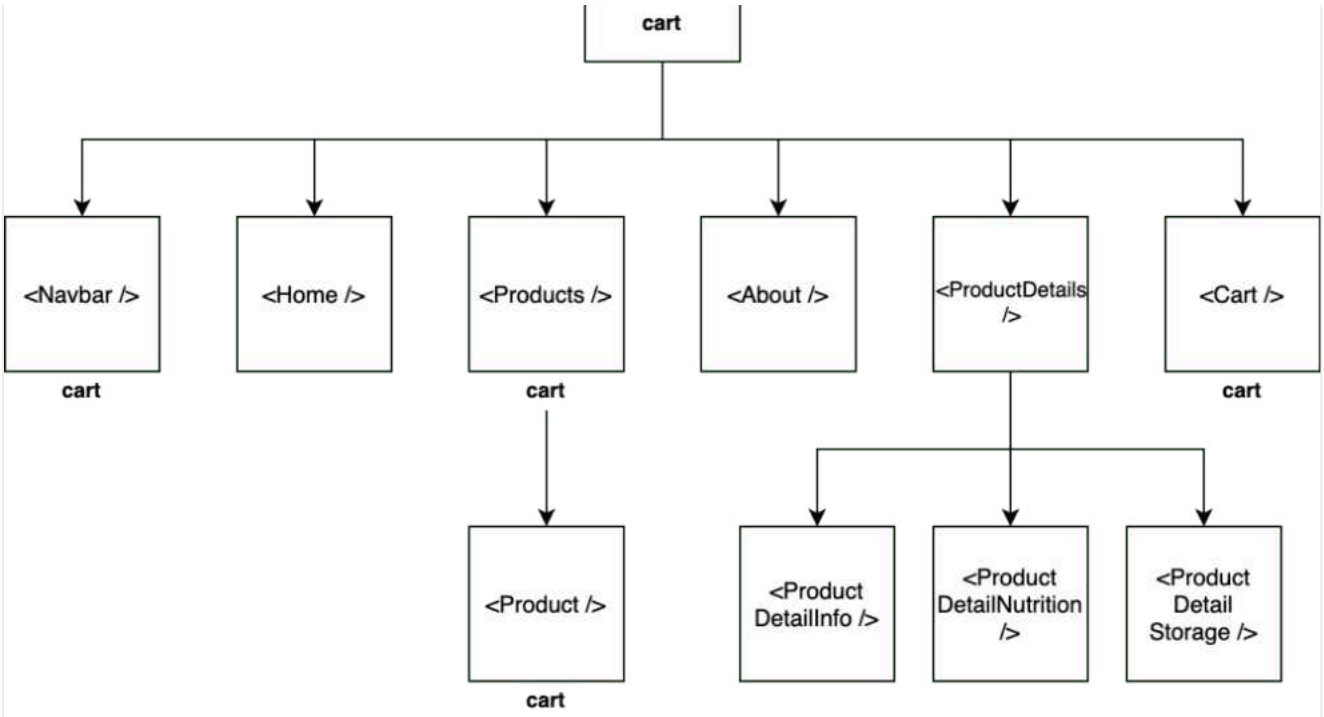
Here's a **tree** representation of the components in our app:



In which component do you think we should have the **cart** state?

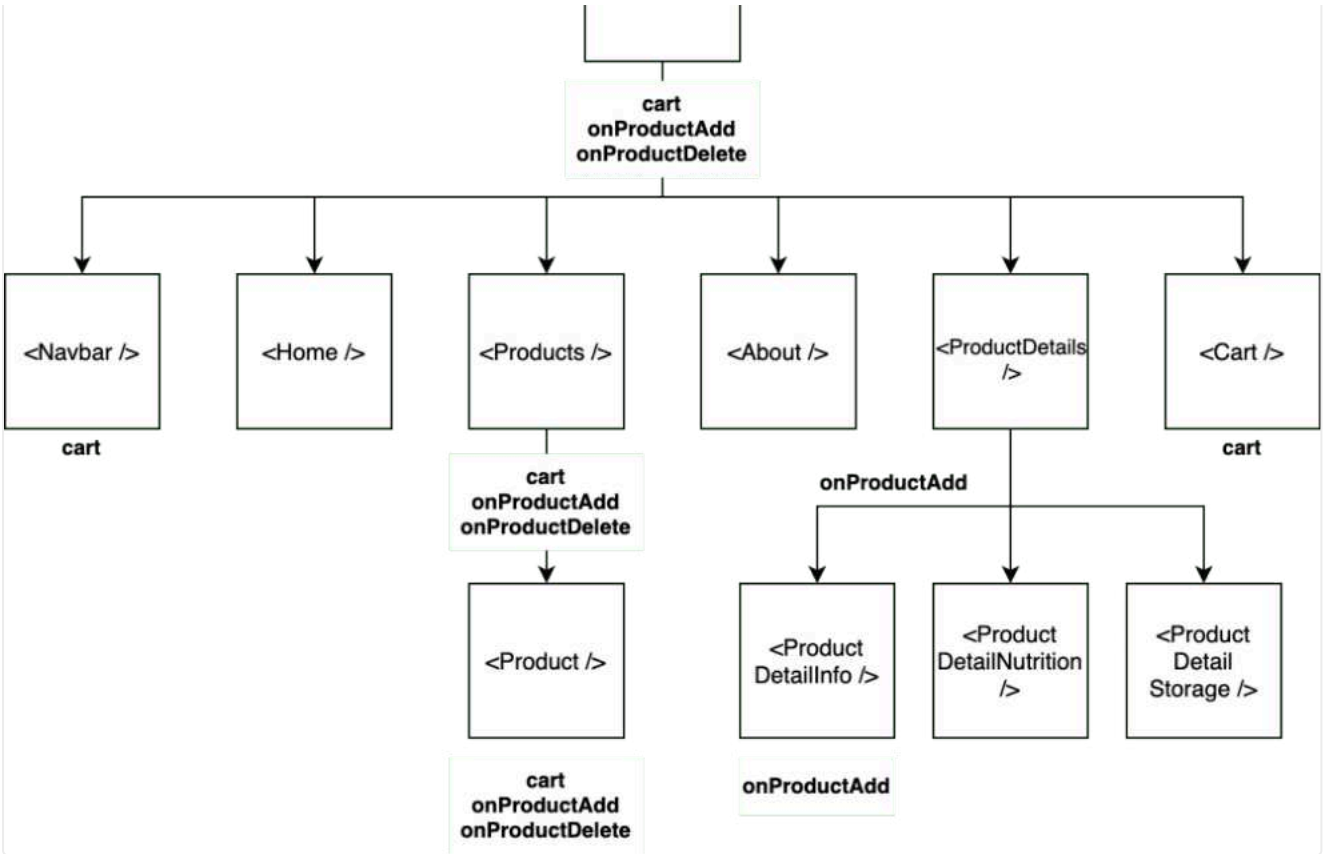
We should lift the state up to the closest common ancestor (Review [Lifting state up lesson](#)).

In this case, this closest common ancestor is the `<App />` component. We'll then pass the `cart` state down as a prop to the components that need it. Here's what that will look like:



Additionally, we'll have 2 methods to manage the `cart` state: `handleProductAdd` and `handleProductDelete`.

These two functions will be passed as `onProductAdd` and `onProductDelete` respectively to follow the naming convention that makes it clear that they are functions. These two functions will also need to be passed down to the components that need them.



Was this helpful?

Add/delete product

Last updated July 2022

In the next project, we'll implement the `handleProductAdd` and `handleProductDelete` functions.

handleProductAdd

When adding a product to the cart, we must store the following information for every product:

- `id`
- `price_id` (so that we can implement Stripe Checkout later on)
- `name`, `price`, `image` so that we can display the basic product details on the cart page
- `quantity` which represents the number of times the client wants to order this product

The cart object starts by being an empty array `[]`. Then let's say the client adds the product with `id=1`, the cart should look like this (some properties are skipped for brevity):

```
[{
    id: 1,
    price_id: "price_XYZ",
    name: "Cheese",
    quantity: 1
}]
```

Then the client adds the same product again to the cart. So we need to update the `quantity` to 2. The cart now looks like the following:

```
[{
    id: 1,
```

```
    quantity: 2
}]
```

Then the client adds the product with `id=2`. Here's how the `cart` will look like this:

```
[{
    id: 1,
    price_id: "price_XYZ",
    name: "Cheese",
    quantity: 2
}, {
    id: 2,
    price_id: "price_XYZ",
    name: "Milk",
    quantity: 1
}]
```

Now, what do you think will happen if the user goes back to add the product with `id=1` again?

We should update the quantity from 2 to 3. We should **not** add the product again to the cart as a new entry because it already exists. The cart should look like this:

```
[{
    id: 1,
    price_id: "price_XYZ",
    name: "Cheese",
    quantity: 3
}, {
    id: 2,
    price_id: "price_XYZ",
    name: "Milk",
    quantity: 1
}]
```

1. Adding a product for the first time (product does not already exist in the cart)
2. Adding a product that was previously added to the cart (so we need to increase the quantity)

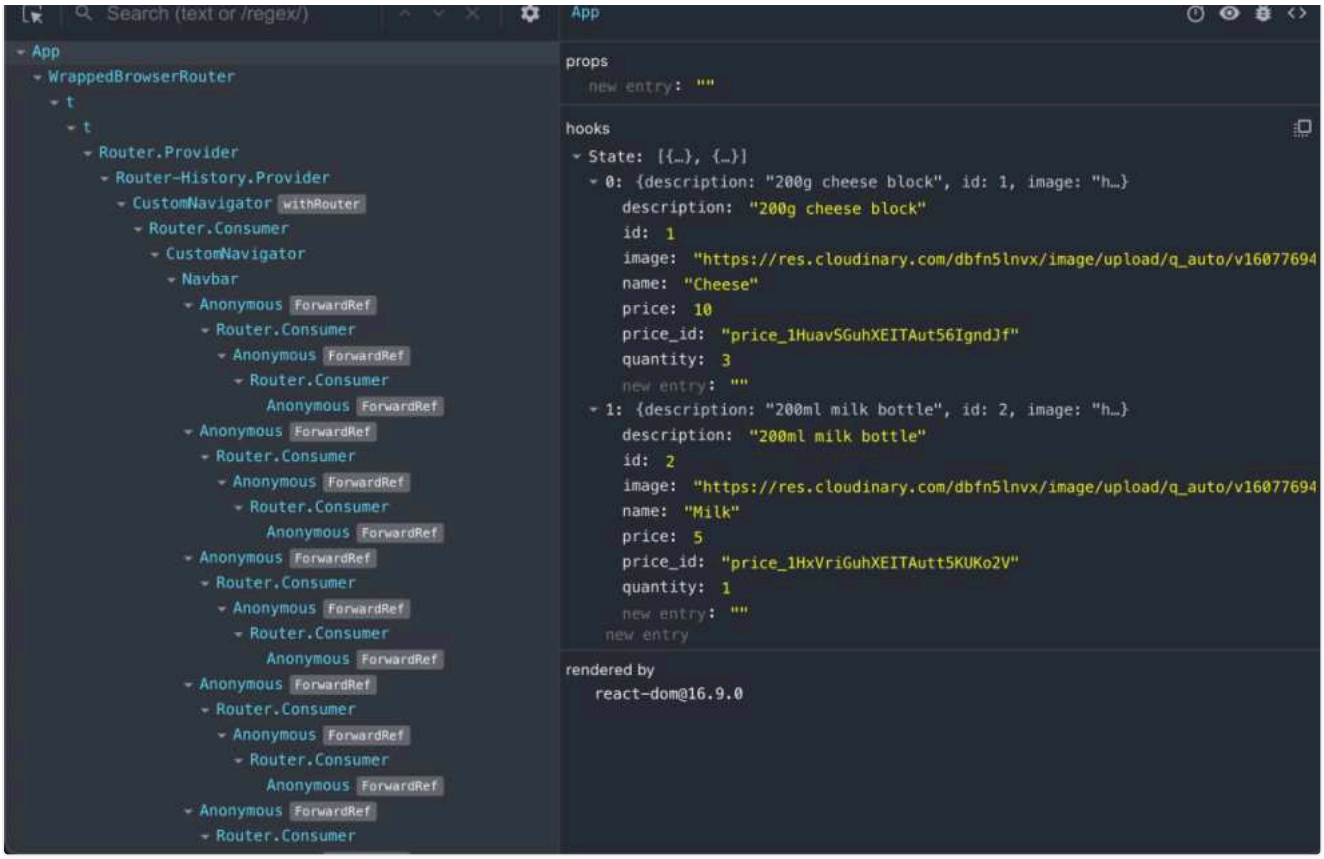
Writing this function is a little bit complicated, especially since state updates need to be **immutable**. So we'll give you hints bit by bit while you're solving the project.

handleProductDelete

Once you've implemented the `handleProductAdd`, you should move on to implement the `handleProductDelete`, which should remove the product completely from the cart. Even if the product was added several times (for example, `quantity=4`), you should remove the product completely from the cart. This kind of update should also be **immutable**.

React dev tools

If you have the React dev tools extension installed, now is a good time to visualize the state by selecting the `<App />` component in the React Dev Tools extension.
You will be able to visualize the state as you interact with your app.

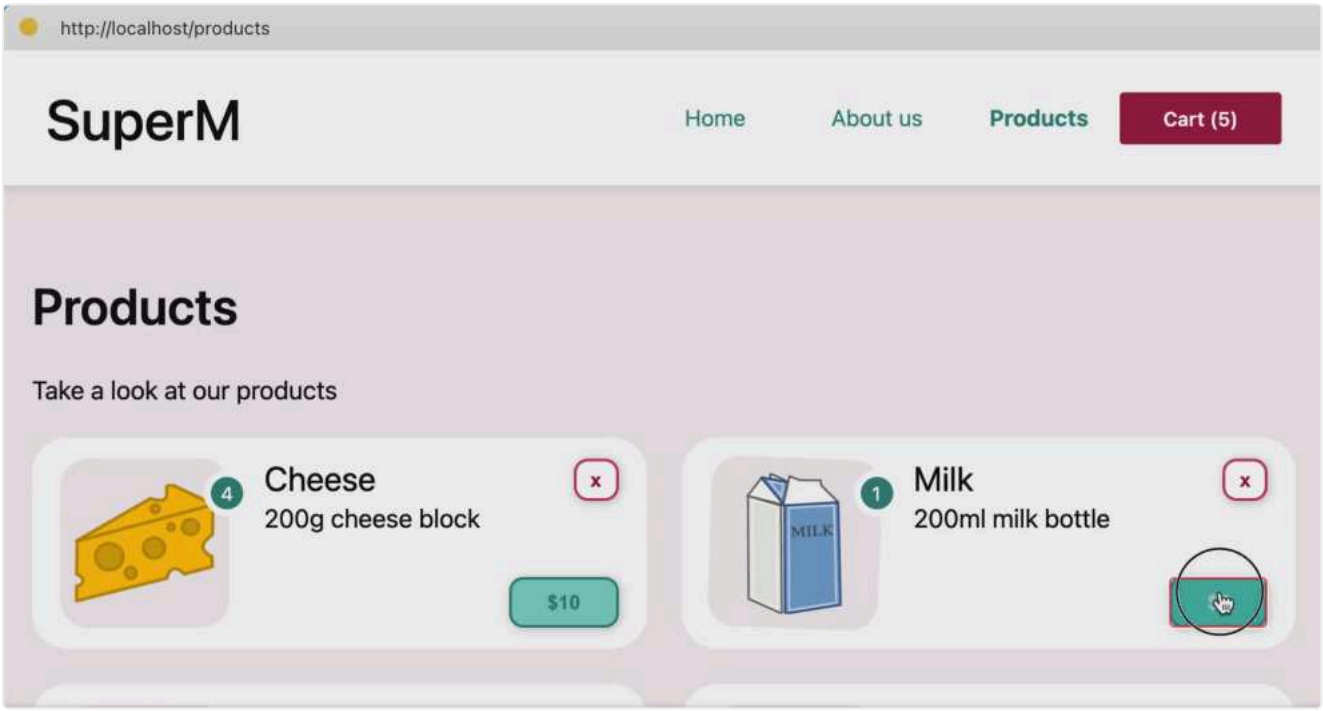


Was this helpful?

Quantity & Cart count

Last updated July 2022

Now that we've implemented the adding and removing to cart functionality, it's time to update the cart count in the navbar as well as show the quantity that the client is ordering on the Products page.



Show product quantity

To show the `quantity` that is being ordered by the client in the Products page, you will need to go through the `cart` (received by the `<Products />` component as a prop) and see if this current product exists in the cart.
If that's the case, you can then pull out the quantity from the cart. Otherwise, we default to 0.

Afterward, we will implement some UI improvements such as hiding (removing from the DOM) the quantity and the delete product button when the quantity is 0.

The `<Navbar />` shows **Cart (0)**, which has been hard-coded for a while now.

The `<Navbar />` component receives the `cart` as a prop, so you will be able to display the actual count of products in between the parentheses.

However, this will **not** be `props.cart.length`, because in the `cart` below:

```
[{
    id: 1,
    price_id: "price_XYZ",
    name: "Cheese",
    quantity: 2
}, {
    id: 2,
    price_id: "price_XYZ",
    name: "Milk",
    quantity: 1
}]
```

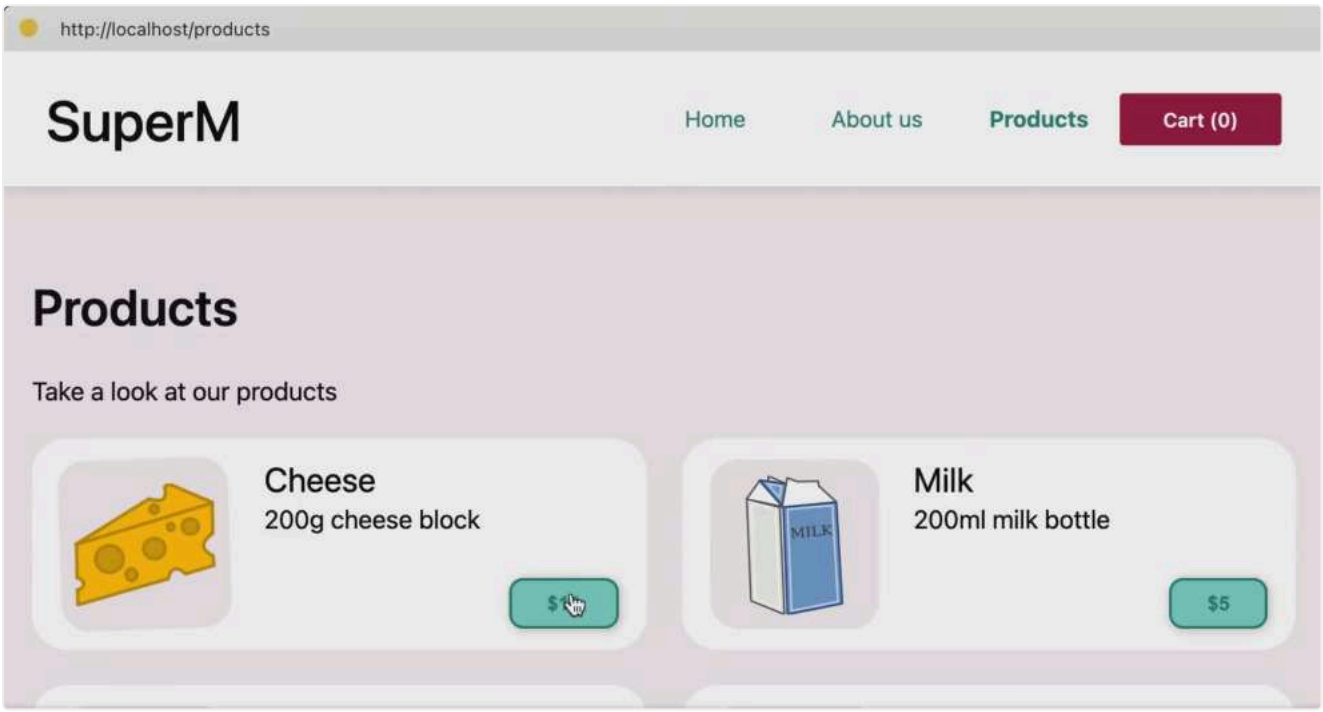
There are three products rather than two (2 x Cheese products and 1 x Milk product).

Was this helpful?

Implementing the cart page

Last updated July 2022

In the next project, you will implement the **cart** page where the client can see a breakdown of the products they ordered and the quantity for each product.



They will also be able to see the **total** price of the cart. This can be calculated by summing the total of (product x quantity) for every product in the cart.

[optional] Restore cart

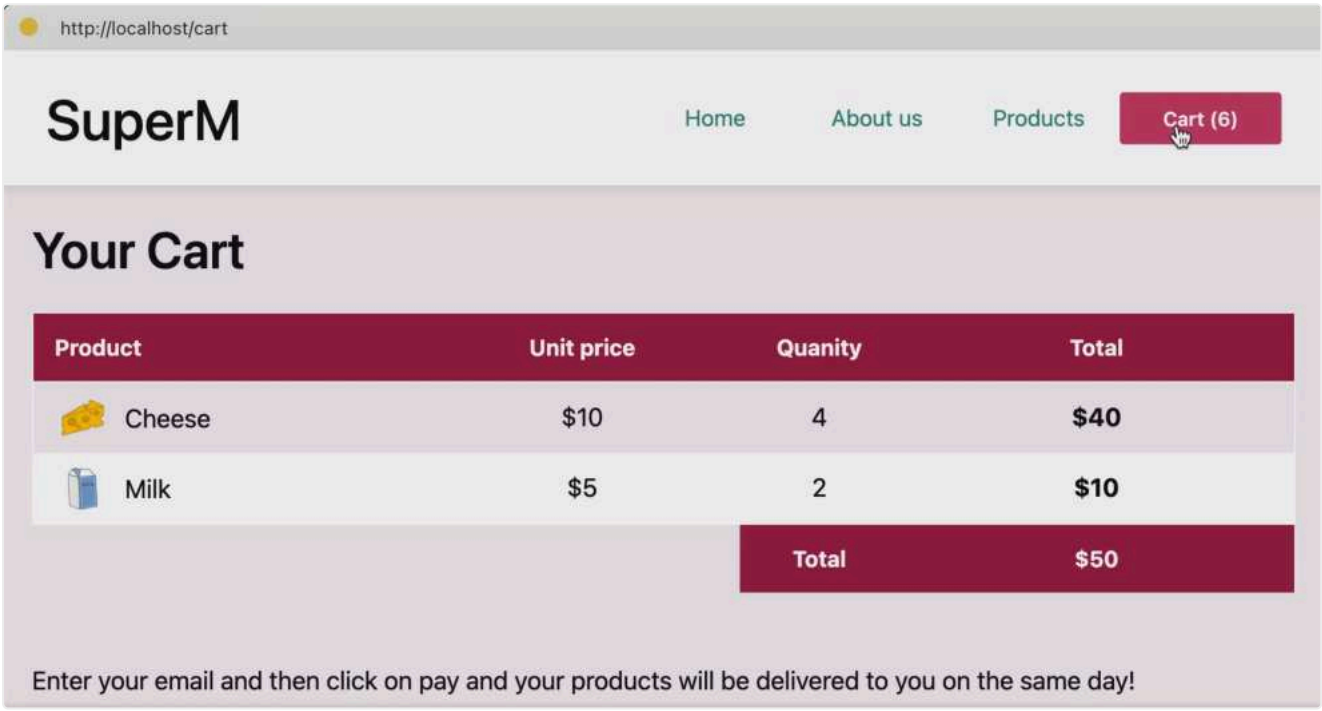
Another part of the project will be implementing the automatic restoration of the client's cart by saving it to localStorage.

For that, you will need to save the client's `cart` (state) to localStorage every time it changes.
And then, when creating the state, you need to try and read it from localStorage (if it exists). Otherwise, default to the empty array.

Integrating Stripe Checkout

Last updated July 2022

The final part of the final project requires implementing Stripe Checkout, which will allow the client to pay for the products they added to their cart. This will create an order on the Stripe Dashboard, which the supermarket can then process.



Back in the [integrating stripe-js lesson](), we called the `stripe.redirectToCheckout` method and gave it the following `lineItems`:

```
lineItems: [{
    price: 'REPLACE_WITH_PRICE_ID_HERE',
    quantity: 1,
}]
```

This `lineItems` accepts an array meaning that you can pass several lineItems representing the products and quantities the client would like to purchase. For example:

```
lineItems: [{
```

```
}, {
    price: 'ANOTHER_PRICE_ID_HERE',
    quantity: 1,
}]
```

This will create a payment request for the *3 x the first product* and *1 x the second product*.

We already created the products in the Stripe Dashboard, and the `price_ids` are provided by the API endpoints that you've used in the previous part of this project. This means that you will find the `price_id` in the `cart` object.

Was this helpful?

Project recap

Last updated May 2023

Congratulations on completing the final project! 🎉🎊✨

You can download a **zip** file containing the solution of the final project below:

- Superm zip solution in Vite + React 18 with useFetch + React Router v6
- Superm zip solution in Vite + React 18 with useSWR + React Router v6

The ZIP file does NOT contain the Stripe test key, as you should be creating your own. Feel free to remix the project. You can use it for your job interview to start your own project. You can add features, remove features, and host the project online. The **only restriction** is hosting the original project with the instructions that were provided on React Tutorial.

Feel free to refer back to the Deploying to Netlify lesson if you'd like to deploy it yourself.

That was a big comprehensive project that included the following major React concepts that you've learned about throughout this course:

- JSX
- Props
- State
- Hooks
- Effect
- Components
- Immutability
- Events
- Forms
- Lifting state up
- Passing props down

- Restoring from localStorage
- React Router
- and many more!

It is an outstanding achievement to complete all of these topics and this final project, so take a moment to celebrate!

Completion certificate

While you have completed the course, we still have some optional chapters for you. You can decide to go through these chapters or skip them. Once you complete the optional chapters, you will have the option to create your completion certificate.

Was this helpful?

Refactoring SuperM to Context

Last updated December 2022

This is your first optional chapter. Optional chapters can be skipped by clicking on the button in the top-right corner.
When you skip a chapter, the last lesson of this chapter will be marked as complete, so you will be able to proceed to the next one.

Not sure whether you should take or skip this chapter? Then read below. Every optional chapter has a short description of when/why you might need it. Then, you can decide for yourself.

Chapter overview

There are several ways to work with state in React:

1. Pass props down (we used this approach in the final project)
2. Using Context (this chapter)
3. Using a state management library (for example, Redux (an intro to Redux is covered in this course), Mobx, etc.)

There isn't a silver bullet. Choosing one of the above depends on the following:

- Your preferences (which approach are you comfortable with)
- Existing approach already in use in an application. If your team is already using Redux, it makes sense to continue using it.
- The team & project size. Context works well for small-to-medium projects/teams.

This chapter will refactor the `SuperM` final project using Context. So, we will start by getting rid of all the passing down props and then use an `AppContext` which will contain our app state as well as the functions that alter the state (`handleProductAdd`, `handleProductDelete`, etc.)

The major downside of using Context is that every state update in the Context will trigger an update across all the components subscribed to that Context (the components that call `useContext()`).

This is why Context is primarily suitable for small to medium sized-apps.

Was this helpful?

Refactoring steps

Skip chapter?

Last updated July 2022

The starting point of the next challenge will be `SuperM` without the Stripe integration and without recovering from localStorage. All the state and state functions will be there (you wrote them in the final project).

The following steps will be provided with more details in the next challenge:

1. Create AppContext

We'll start by creating a Context called `AppContext`. This context will contain the `cart` state as well as all the functions that update the `cart` state.

So, we will move the `handleProductAdd` and `handleProductDelete` functions from the `App.js` into the `AppContext`.
The context should return the `cart` and the 2 methods: `onProductAdd` and `onProductDelete` (which correspond respectively to `handleProductAdd` and `handleProductDelete`).

2. Wrap app with AppContext

We'll need to wrap our app with the `<AppContext />` so that its child components can read its context. Also, every component that needs to use the `AppContext`, needs to import `useContext` and use that hook at the top of the component.

3. Update components

We'll need to update our components to use the `cart` state from the `AppContext`.
We'll have to do the same for the `onProductAdd` and `onProductDelete`.

4. Taking it a step further

was in the `<Navbar />`, `<ProductDetails />`, and `<Cart />` components.

We'll extract it into reusable functions such as `getTotalPrice()`, `getProductFromCart()`, and `getCartCount()`.

This will have several benefits:

1. The components will be slimmer.
2. These functions can be re-used if we ever need them in another component.

Was this helpful?

Dark theme

Skip chapter?

Last updated December 2022

We'll be adding a dark theme to our SuperM app. The solution will mostly be written in CSS as it's always better to leverage platform features whenever possible. This is in line with what we recommended back in the **Styles in React lesson** lesson.

Throughout this course, we had several examples where `isDarkTheme = true` or `theme = dark` was passed as a prop to a component. While this is possible, it's often possible to implement dark mode entirely from CSS.

You will only need to write a couple of lines of code to add the class `dark` to the `<body>` or `<html>` element from within React, and the rest is handled by CSS.

That's because CSS Variables can be overridden. Here's the gist of it:

```
:root {
    --bg-color: white;
    --text-color: black;
}

body {
    background-color: var(--bg-color);
    color: var(--text-color);
}

body.dark {
    --bg-color: #222; /* almost black */
    --text-color: white;
}
```

Notice how we define two CSS variables, `--bg-color` and `--text-color`, and then set the `background-color` and `color` properties on the body to use the values of

Then, when the body has the class dark (body.dark), we change the values of these variables and flip them around. So, the background color becomes dark, and the text color becomes white.

This works most of the time; however, you will often have to make minor adjustments for borders, box shadows, etc.

Default to OS Preference

This section will be optional in the next challenge. We use this approach on the main app that React Tutorial runs. We default the theme based on your OS (Operating System) preference.

```
const prefersDark = window.matchMedia("(prefers-color-scheme:
dark)").matches;
console.log(prefersDark); // either true or false (depending on
your OS preference)
```

Based on this variable, you can then add the class `dark` to the body (to enable dark theme) or not.

You can read more about it on web.dev/prefers-color-scheme.

Was this helpful?

Intro to class components

Skip chapter?

Last updated March 2023

The goal of this optional chapter is to explain class components in React. You already know all the React concepts using hooks, and this chapter will explain their class equivalent.

Not sure if you should complete or skip this chapter? Then read the chapter overview below.

Chapter overview

React started with class components (they even started with React.createClass, but this is way too old now) and then announced hooks in 2019.

Class components are not going anywhere; however, everyone seems to find it easier to write hooks than classes.

If you're working on a big project with several team members that was started a while ago, you will most likely encounter several class components, which is why it would be useful for you to go through this chapter.

On the other hand, if you're working with React on a modern project or by yourself, then you can do everything with hooks without classes. In that case, it might still be useful to go through this chapter to understand the syntax of classes as some documentation and Q&A is written in classes.

All the main concepts we learned in this course can be done in classes. We'll cover the following:

- Components
- Props
- State
- Events

At the moment, there's only one feature supported in classes but not in hooks, which is error boundaries. We have a video screencast on Error Boundaries at the end of this chapter.

Why did developers find classes more complicated?

Most developers found classes to be more complicated because they required you to have a deep understanding of how classes work in JavaScript.

We'll cover some of the most common bugs developers encounter when working with classes in React (such as overriding the constructor and forgetting to call super, binding functions in events, etc.)

Was this helpful?

Class Components

Skip chapter?

Last updated December 2023

The following function component:

```
import {createRoot} from "react-dom/client";

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

createRoot(document.querySelector("#react-root")).render(<App
/>);
```

Can be written as a class:

```
import React from "react";
import {createRoot} from "react-dom/client";

class App extends React.Component {
    render() {
        return (<h1>Hello World</h1>);
    }
}

createRoot(document.querySelector("#react-root")).render(<App
/>);
```

Notice how the react-dom bit does not change at all. The only thing that changes is instead of defining a function; we have to define a class now.

The name of the class determines the component. In this example, class App creates the component <App />.

automatically whenever it needs to render the component to the DOM.

So, your JSX from the function goes straight into the `render()` method. Everything you know about ASI, returning `null` when you don't want to return anything from the component, etc., still applies over here.

Class ... extends React.component

There's one more detail left: your class **must** extend (inherit) from `React.Component`.

This will allow your component to inherit all the necessary functionality from the React library.

If you find the syntax `React.Component` confusing, here's why it looks like this:

- `React` is an object.
- `Component` is a property on the `React` object that you access with `React.Component`.
- `React.Component` is a class which is why we're able to inherit from it.

You can also, if you'd like, import the `Component` named export directly, then your code becomes as follows:

```
import {Component} from "react"; // this changed
import {createRoot} from "react-dom/client";

class App extends Component { // this changed
    render() {
        return (<h1>Hello World</h1>);
    }
}

createRoot(document.querySelector("#react-root")).render(<App
/>);
```

Recap

- The name of the class determines the name of the component.
- Your class must extend from `React.Component`.
- Your class must have a `render()` method that returns the JSX for that component.

Was this helpful?

Props

Skip chapter?

Last updated March 2023

In class components, the `props` are received by the `React.Component` class constructor and captured as instance variables `this.props`.

So, if you want to access the props from anywhere inside the component, you'll need to access them with `this.props`. For example:

```
import React from "react";

class App extends React.Component {
    render() {
        console.log(this.props); // {size: "large"}
        return (<h1>Hello World</h1>);
    }
}

const element = <App size="large" />;
```

Notice how you can access the props anywhere in the class with `this.props`.

Overriding the constructor (old way)

Later in this chapter, we'll need to override the constructor to handle events. For that, you need to be careful and make sure to pass `props` to the parent constructor with `super(props)`.

Since your class is extending `React.Component`, and since you're overriding the `constructor()`, this means that the `React.Component`'s `constructor()` will not execute anymore.
So, you will have to call `super(props)`, which will pass the `props` to `React.Component`'s constructor. Here's what the syntax for it looks like:

```
class App extends React.Component {
    constructor(props) {
        super(props); // has to be the first line inside the
constructor
        // the rest of your constructor code can go here
        this.state = {};
    }

    render() {
        console.log(this.props); // { ... }
        return (<h1>Hello World</h1>);
    }
}
```

Notice how your `constructor()` receives the `props` and then passes them to the parent class's (`React.Component`) constructor with `super(props)`.
If you've never seen `super` before, this is a JavaScript keyword that allows you to access methods on the parent class.

`super(props)` has to be the first line inside your constructor. And then, inside of that constructor, you can access the props with `this.props` or `props`. Remember never to modify the props.

> Note that the text editor on React Tutorial might strike through the `super(props)` call and say that it's deprecated. Please note that this is a text editor bug, and `super(props)` is not deprecated.

Using public class fields (new way)

We recommend that you use this newer syntax as it's easier. You neither have to override the constructor nor call super anymore.

Using JavaScript's new *public class fields* syntax, you can define your state variable using the following syntax:

```
class App extends React.Component {
    // public class fields
    state = {};

    render() {
        console.log(this.props); // { ... }
        return (<h1>Hello World</h1>);
    }
}
```

We'll explain how you can use this syntax to set the state in the following lesson.

Recap

- You can access the props anywhere in a class component with `this.props`.
- If you need to override the constructor on your class component, you will need to call `super(props)` first thing.

Was this helpful?

State

Skip chapter?

Last updated December 2023

With class components, state is declared as an instance variable called `state`. Thus, you can access it with `this.state`.

So, the following function component:

```
import {useState} from "react";

function Button() {
    const [isDisabled, setIsDisabled] = useState(false);

    return <button disabled={isDisabled}>Button text</button>;
}
```

can be re-written as a class component (old syntax):

```
class Button extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            isDisabled: false
        };
    }

    render() {
        return <button disabled={this.state.isDisabled}>Button
text</button>;
    }
}
```

Notice how we had to define the state with `this.state = {isDisabled: false}` in the `constructor`. And, then, you can read this state with `this.state.isDisabled`.

```
false};

class Button extends React.Component {
    state = {
        isDisabled: false
    };

    render() {
        return <button disabled={this.state.isDisabled}>Button
text</button>;
    }
}
```

Updating state

Just like with hooks, directly updating the state is not allowed. You have to use a dedicated method. In class components, you get a `this.setState()` instance method that is inherited from `React.Component`.

Once we learn about events in the next lesson, we'll learn more about updating the state with `this.state()`.

Multiple state

It's also possible to have several state in the same class components. The `state` instance variable is an object, so you can add more `key/value` props. For example:

```
// inside the constructor
this.state = {
    isDisabled: false,
    grades: [10, 20],
    counter: 0
}
```

`this.setState({isDisabled: true});` This will override the previous value of `isDisabled` but it won't affect the value of `grades` or `counter`.

State updates must be immutable

Everything you learned about immutability still holds true.
State updates have to be immutable for React to know that the value has changed while using the `===` comparison.

Multiple state updates may be batched

Just like with `useState`'s set state function, multiple state updates may be merged. So, whenever you need the previous state value to calculate the new one, you will have to use a function:

```
this.setState((prevState, props) => ({
  counter: prevState.counter + 1
}));
```

The function will receive the previous state as well as the props.

Recap

- State is defined as an instance variable. You can access it with `this.state`.
- `this.state` is an object that can contain one or more state. Each state is a `key/value` pair in the `this.state` object.
- You can access the state with `this.state.keyNameHere`.
- However, you should not update the state directly. You have to use `this.setState({key: newValue})`
- State updates are merged. If you have multiple states, you can update only one key, and it won't affect the other ones.

Was this helpful?

Events

Skip chapter?

Last updated December 2023

When working with Class Components, events become a little bit trickier due to the context of `this`.

Let's start by seeing how we can re-write an `onClick` handler from a function component into a class component:

```
function App() {
    function handleButtonClick() {
        console.log("Button clicked");
    }

    return <button onClick={handleButtonClick}>Add 1</button>;
}
```

We can re-write it as a Class Component:

```
import React from "react";

class App extends React.Component {
    handleButtonClick() {
        console.log("Button clicked");
    }

    render() {
        return <button onClick={this.handleButtonClick}>Add
1</button>;
    }
}
```

Notice how we created an instance method `handleButtonClick` and then changed the `onClick={handleButtonClick}` to `onClick={this.handleButtonClick}`.

handleButtonClick, you will get an error saying that setState does not exist on undefined.

So, even if you try to access `this.props`, you will get an error saying that `props` does not exist on `undefined`.

This is because `this` inside of the `handleButtonClick` is not pointing to the current instance of the class anymore. Instead, it has a value of `undefined`.

This happens because of how `this` works in JavaScript. Specifically, in React's use case, React is calling the `handleButtonClick` function from outside of the class (when the user clicks on the button) thus the context of `this` is lost so `this` has a value of `undefined` now which is not useful.

There are several ways to fix this issue. We'll highlight the two main ones that you can use:

1. Bind this in the constructor

In order to fix this issue, you can override the `constructor` and then **bind** `this` to the event handler that you created:

```
import React from "react";

class App extends React.Component {
    constructor(props) {
        super(props);
        // bind this
        this.handleButtonClick =
this.handleButtonClick.bind(this);
    }

    handleButtonClick() {
        console.log("Button clicked");
    }

    render() {
        return <button onClick={this.handleButtonClick}>Add
```

```
}
```

The `.bind(this)` is passing the current context of the class (the current instance) to `handleButtonClick` so that it's not lost when React calls your event handler.

The upside of this method is that if you have several `onClick={this.handleButtonClick}`, all of them will work since `handleButtonClick` will receive the correct value of `this`.

The downside of this method is that it's a bit lengthy, especially when you have several event handlers. Also, you are forced to override the `constructor`, which means you have to remember to receive the `props` as an argument and then call `super(props)`.

You can also re-write this using the newer syntax. However, both options are unfortunately not very nice to read:

```
import React from "react";

class App extends React.Component {
    // bind this
    handleButtonClick = this.handleButtonClick.bind(this);

    handleButtonClick() {
        console.log("Button clicked");
    }

    render() {
        return <button onClick={this.handleButtonClick}>Add
1</button>;
    }
}
```

2. Arrow function (lexical context)

Another method would be using arrow functions since they have a lexical scope. This means that they do not have their own value of `this`, and, instead, they persist the

```
import React from "react";

class App extends React.Component {
    handleButtonClick() {
        console.log("Button clicked");
    }

    render() {
        // use arrow function for event handler
        return <button onClick={() =>
this.handleButtonClick()}>Add 1</button>;
    }
}
```

Note the `() => this.handleButtonClick()` instead of `this.handleButtonClick`. Don't forget the `()` after `handleButtonClick`; that's because we're now passing a function definition. So, inside this function definition, you have to call `handleButtonClick()`.

The upside of this technique is that it's short and easy to use. You don't need to override the `constructor`.

The downside of this technique is that if you have several `onClick={this.handleButtonClick}`, you will need to update all of them to `onClick={() => this.handleButtonClick()}`.

In summary, this is one of the examples that shows one of the reasons why React has tried to move into function components with hooks. The `this` concept is often confusing for a lot of developers. But, as you saw with hooks, we never had to deal with this issue.

You will notice in the next challenge that the component written with *hooks* is much smaller than the one written with *class*.

Recap

- In order to use `this` inside an event handler (for example, `this.setState` or `this.props`), you have to either:
- 1. bind the value of `this` in the constructor
- 2. or use an arrow function to maintain the lexical scope.

Was this helpful?

Component lifecycle

Skip chapter?

Last updated February 2023

We learned in this course about the `useEffect` hook that allows us to run a piece of code at a specific lifecycle of the component.

Here's a refresher of the most common use cases of `useEffect`:

```
import {useEffect} from "react";

function App() {
    useEffect(() => {
        console.log("first render");
        return () => {
            console.log("will be removed");
        }
    }, []);

    useEffect(() => {
        console.log("first render + subsequent updates")
    });
}
```

When using Class Components, React allows you to define specific instance methods that will be automatically called by React.

There are three common methods:

- `componentDidMount`: runs the first time the component is rendered to the DOM.
- `componentDidUpdate`: runs every time the component has updated (but not after the first render).
- `componentWillUnmount`: runs before the component is being removed from the DOM.

Here's an example:

```
class App extends React.Component {
    componentDidMount() {
        console.log("mounted");
    }

    componentDidUpdate() {
        console.log("updated");
    }

    componentWillUnmount() {
        console.log("unmounting");
    }

    handleClick() {
        // a way to re-render the component
        this.setState({date: new Date()});
    }

    render() {
        return <button onClick={() => this.handleClick()}>Re-
render</button>;
    }
}
```

The component above will log `mounted`. Then, every time you click on the button, it will log `updated`. Finally, if/when the component is being removed from the DOM, it will log `unmounting`.
So, React is automatically calling these instance methods if you define them. You don't have to define all of them. You can define only the methods that you need.

Difference with useEffect

There's a striking difference between lifecycle hooks with useEffect compared to lifecycle with class components:

```
useEffect(() => {
```

The above `useEffect` runs on the first render as well as on the updates. However, with class components, these two lifecycles are separate. You have to use both `componentDidMount` and `componentDidUpdate`. For example, if you needed to log the current date on the first render and subsequent updates, you'd have to write the following:

```
componentDidMount() {
    console.log(new Date());
}

componentDidUpdate() {
    console.log(new Date());
}
```

Historically, this often leads to a lot of code duplication, which is why the React team has decided to merge these two lifecycles into 1 when using `useEffect`.

Equivalent to effect dependencies

The `useEffect` had an array of dependencies that allowed you to specify when the effect should run. For example, a dependency of `[count]` would only run the effect when the `count` state changes:

```
import {useState} from "react";

function App() {
    const [count, setCount] = useState(0);

    useEffect(() => {
        console.log("Count changed");
    }, [count]);
}
```

adding an if condition inside a componentDidUpdate method and comparing the previous state to the current state. This is why the `componentDidUpdate` method receives the `prevProps` and `prevState` as arguments:

```
componentDidUpdate(prevProps, prevState) {
    if (prevState.count !== this.state.count) {
        console.log("Count changed");
    }
}
```

It's also worth noting that if you want to have the exact behavior as the `useEffect` code, then you will have to add a `componentDidMount` for the same reason that was mentioned in the *Difference with useEffect* section.

Recap

- There are three common lifecycle methods when using Class Components:
- `componentDidMount`: runs the first time the component is rendered to the DOM.
- `componentDidUpdate`: runs every time the component has updated (but not after the first render).
- `componentWillUnmount`: runs before the component is being removed from the DOM.

Was this helpful?

Intro to Preact

Skip chapter?

Last updated January 2023

The goal of this optional chapter is to give you an introduction to **Preact**.

Not sure if you should complete or skip this chapter? Then read the chapter overview below.

Chapter overview

Back in the **React intro > Install & setup** and the **React DOM > ReactDOM Usage** lessons, we saw that React weighs approximately 6KB while ReactDOM weighs approximately 130KB. Both combined would approximate 136KBs.

This measures the minified code that gets executed by the browser. You may often hear people referring to the size when it's *GZipped*; for `react` + `react-dom`, that would be approximately 42KBs.

Gzip vs. Minified

Gzip is a compression algorithm that compresses the JavaScript code before sending it from the server to the browser. However, you should remember that while the Gzipped file is transferred (for example, 45kbs are transferred from the server to the browser), the browser then has to uncompress the Gzipped version. So, the browser is not parsing and executing 42KB but approximately 130KB of minified JavaScript code.

In summary, a JavaScript file containing React + ReactDOM minified costs 130KB. When your users browse your website, the server compresses this file down to 42KB using Gzip. So, the browser has to download 42KB of JavaScript.
After the file has been downloaded, it has to uncompress it, and the file goes back to being 130KB. This is the code that the browser ends up parsing, compiling, and executing.

While `React` + `react-dom` cost 130KB minified (42KB Gzipped), Preact is a much more lightweight alternative.

Preact is approximately 10KB minified (4kb gzipped). **Bundlephobia for Preact**. Sometimes, Preact by itself is not enough and you may have to use `preact-compat`, which costs 14KB (and 5KB Gzipped). **Bundlephobia for preact-compat**.

In total, `preact` + `preact-compat` weigh approximately 24KB (9KB Gzipped).

We'll explain the purpose of `preact-compat` in a later lesson in this chapter.

When to use Preact?

While the difference is big, you might be wondering whether this matters for the performance of your website. The answer is **it depends**.

It depends on your target market and the kind of device that they own. It depends on how much JavaScript you're already shipping. If you're shipping 2MB of uncompressed JavaScript (you shouldn't), then a reduction of 100KB won't have that big of a difference.

The most important factor, however, is your target audience. I've worked with several companies on Web Performance, and I could see how depending on their target markets, they could "get away" shipping more JavaScript to the browser.

On the other hand, *React Tutorial* has students all over the world, so I'm always prioritizing performance to make sure it's accessible to as many people as possible.

In summary, if your target market mostly uses laptops or the latest flagship phones, you probably don't need to worry about the size difference between React and Preact.

With these kinds of decisions, it's important to measure the performance of your website before and after such change to be able to determine how much of an impact it brought.

Another thing you need to consider is that not all libraries/frameworks are compatible with Preact, so you should probably take a look at that.

If you're building your first website ever, I would continue using React because it would be easier for you to fix bugs you encounter. However, I'd still go through this chapter as Preact experience is a nice thing to mention during an interview.

Is Preact a 1-to-1 replacement for React?

From the Preact documentation:

> Preact itself is not intended to be a reimplementation of React. There are differences. Many of these differences are trivial or can be completely removed using `preact-compat`, a thin layer over Preact that attempts to achieve 100% compatibility with React.

In the next lessons and challenges, we'll show you common use cases of Preact. One of them where you have to use `preact-compat` and the other one where you don't.

Was this helpful?

Preact from scratch

Skip chapter?

Last updated July 2022

There are two main use cases if you decide to work with Preact:

1. Creating an app from scratch using Preact.
2. Migrating an existing app from React to Preact.

In this lesson, we'll cover the first use case.

Installation

The recommended way to start a new Preact project is using the `preact-cli`:

```
npm install -g preact-cli
```

Then `cd` into a folder of your choice and run the `preact create` command:

```
preact create default my-project
```

You can replace `my-project` with your project name. This uses the `default` template; however, there are other templates that you can list with `preact list`.

[Preact Getting started docs]

JSX and h()

If you then navigate to `src/components/app.js`, you will notice that the code looks quite familiar. There are only two differences:

```
import { h } from 'preact';
import { Router } from 'preact-router'
```

enough for what you're looking for, you can use `preact-loader` with `preact/compat`. We'll cover `preact/compat` in the next lesson.

2. We're importing an h function from `preact`, which doesn't apparently seem to be used anywhere in that file. This `h()` function is responsible for converting your JSX code into Preact elements (similarly to how you had to import `React` to make JSX work). So, the `h()` function is similar to `React.createElement`.

Note: the `h()` method is not required in modern Preact projects and might be removed from the boilerplate in the future.

Example component

Let's take a look at a full example Preact component that is rendered to the DOM:

```
import { h, render } from "preact";

const App = () => (
  <div>
    <h1>Preact App</h1>
  </div>
);

render(<App />, document.querySelector("#react-root"));
```

As you can see, the `render` method is also exported from the same `preact` package.

Was this helpful?

Migrating an existing app

Skip chapter?

Last updated July 2023

This lesson covers the second use case for Preact: Migrating an existing app from React to Preact.

In this case, there's a high probability that you're using some react features that are not part of `preact`.

This is where `preact-compat` shines in. It aims to be a drop-in replacement for React.

For large projects, you can instruct `webpack` to resolve the `react` package via `preact-compat`.

This means that whenever `webpack` sees the following imports:

```
import React, {useState, useEffect} from "react";
import {createRoot} from "react-dom/client";
```

Instead of importing these objects and functions from the `react` package, it will import them from the `preact-compat` package.

To make this work, you need to configure `webpack` to swap these packages (the option is called `resolve`):

And inside your `webpack.config.js`:

```
{
    // ...
    resolve: {
        alias: {
            'react': 'preact/compat',
            'react-dom': 'preact/compat',
        }
    }
```

More details can be found on the **official documentation**.

Was this helpful?

React Lazy

Skip chapter?

Last updated July 2022

This optional chapter aims to teach you about `React.lazy`, which allows you to code-split components in your application for better performance.

Not sure if you should complete or skip this chapter? Then read the chapter overview below.

Chapter overview

As your application grows, the number of components will grow.

If you keep on using `import` to import your components, the bundle size of your application will become quite big. This will slow down the initial load of your application.

For example, let's say you've got the following components:

- `<App />`
- `<Sidebar />`
- `<SupportChat />`
- `<AdminChart />`.

Most users need the `<App />` and `<Sidebar />`. Only a portion of them needs the `<SupportChat />`, and another tiny portion needs the `<AdminChart />`. So, why make every user of your app download all of these components and "pay" the download, parse, and compile cost for them if they're not going to use them?

This is where React lazy allows you to lazy load (only load at a later stage, when needed) some of your components.

This optional chapter will start with a `dynamic import` recap from the Learn JavaScript course and then explain how React.lazy can be used in your app.

Dynamic imports

Skip chapter?

Last updated July 2022

This lesson is taken from [Learn JavaScript](#).

Whenever you `import` a file in your JavaScript code, the `import` will happen synchronously. This is called a static import. Let's take an example:

```
import debounce from "lodash-es";
import Helpers, {getDate} from "./helpers.js";

console.log(debounce); // lodash's debounce function
console.log(Helpers); // a Helpers class exported by ./helpers.js
(default export)
console.log(getDate); // a getDate function exported by
./helpers.js (named export)
```

The imports are processed synchronously, meaning the browser will wait until the file has been imported before running the rest of the code. This is the case when you use `<script type="module"></script>` without using a module bundler.

When you use a module bundler, the `import` statements will end up merging the code from the imported files into one main file. So, in the case of `webpack`, you will end up with a `bundle.js` file that contains all of the `debounce` code, the `NumericHelper`, and the `console.log(debounce, NumericHelper)`.

As your application grows, the size of your JavaScript bundle becomes larger and larger, making the initial start-up of your website slower and slower.

Also, sometimes, an import is only needed in a very specific case. For example, let's say you need to load an image upload library *if* the user visits their settings page and clicks on `Edit avatar`. And, it turns out that this is not a very common feature of your website. So, why make all of your users pay for the performance cost of including this library?

When the dynamic `import()` function is executed.

To make an import dynamic, you have to use `import()` as a function.

Additional example

Before we dive into the code, this course heavily relies on dynamic imports. We only load the help center and external library when you click on the *Help* button the first time. It's a big library, and not everybody uses it. This is why we dynamically import it.

import()

Here's how we convert the previous two `import` statements into dynamic imports:

```
import("lodash-es").then(module => {
    const debounce = module.default; // because debounce is a
default export
    // use debounce inside this function...
});

import("./helpers.js").then(module => {
    const Helpers = module.default; // because Helpers is a
default export
    const getDate = module.getDate; // because getDate is a named
export
    // use Helpers and getDate inside this function...
});
```

As you can see, dynamic imports work for libraries and files alike. When you use `import()` as a function (dynamic import), it will return a promise.
When the promise fulfills, it will resolve with a `module` object that contains the exported data.

Default exports are available under the `default` key, and other data is available as keys under the same name they were exported with.

The flexibility of dynamic imports allows you to perform an import whenever necessary. For example, we load the `chat` library when the user clicks on the **Contact support** button. Here's an example of how you can perform a dynamic import when the user clicks on a button:

```
const button = document.querySelector("#button");

button.addEventListener("click", () => {
    import("chat-library").then(module => {
        module.init(); // assuming the chat-library exports a
named function called init.
    });
});
```

This import on user interaction would not have been possible when `import` is used statically (the one we learned about earlier in this course). So, the code below does **not** work:

```
const button = document.querySelector("#button");

button.addEventListener("click", () => {
    // ❌ this does NOT work ❌
    import {init} from "chat-library"
    module.init();
});
```

Example with async/await

Since `import()` returns a promise, you can use `async/await` syntax. Don't forget to make the parent function `async`:

```
const button = document.querySelector("#button");

button.addEventListener("click", async () => {
```

```
});
```

Example with destructuring

You can also use object destructuring here since it's returning an object. For example, we can destructure the `init` function from the `module` object:

```
const button = document.querySelector("#button");

button.addEventListener("click", async () => {
    const {init} = await import("chat-library");
    init();
});
```

Handle errors

Don't forget to handle errors! The dynamic import could fail due to network conditions or other issues. So, don't forget to wrap your `async/await` with a `try ... catch` call or add a `.catch()` to your promise call.

Recap

- Whenever you `import` a file in your JavaScript code, the `import` will happen synchronously. This is called a static import.
- You can dynamically import a file or a library by using `import()` as a function.
- `import()` returns a promise. It resolves with an object containing the exported data.
- `default` exports are available under the `default` key, and other data is available as keys under the same name they were exported with.

React.lazy & Suspense

Skip chapter?

Last updated July 2022

Let's say you've got an `<Admin />` component exported from a local `Admin.js` file that is only rendered when the logged-in user is an admin:

```
import {createRoot} from "react-dom/client";
import Admin from "./Admin.js";

function User(props) {
    return (<>
        <p>User details show up here</p>
        {props.isAdmin && <Admin />}
    </>);
}

// Sample usage
// <User isAdmin={true} /> or <User isAdmin={false} />
```

Notice how the `<p>User details show up here</p>` is always rendered. However, only when the `props.isAdmin` has a value of `true` then the `<Admin />` component is rendered.

Assuming the `<Admin />` component is a very large component (that loads several libraries) and is only used by a small portion of the users, it is wise to lazy load it (only load it when it's used).
This will improve the performance of your app as most users do not need to download all the JavaScript of the `<Admin />` component (and its dependencies).

This can be done in 3 steps:

Step 1: Static import to dynamic import with React.lazy

As long as you have a static import, the code of the `<Admin />` component will end up in your main bundle, as explained in the previous lesson. So, we need to change the

```
import Admin from "./Admin.js";
```

to:

```
const Admin = React.lazy(function() {
    return import("./Admin.js");
});
```

The static import is now a dynamic import: `import("./Admin.js")`.

This dynamic import returns a promise. `React.lazy` is a function provided by React (don't forget to import `React`, but it's probably already imported). This function controls **when** the component needs to be imported. So, even though you see a dynamic import, React **won't** import the component *just yet*.

And finally, we assign the result of `React.lazy` to a variable called `Admin`.

It's important that `React.lazy` receives a function definition that, when invoked, returns the promise of `import()`. So, don't forget the `return` keyword. It's very important.

Alternatively, you can write the code above with an arrow function with implicit return. Then, you can drop the `return` keyword because it's implicit:

```
const Admin = React.lazy(() => import("./Admin.js"));
```

Step 2: Wrap with Suspense

Now that we've converted the import into a `React.lazy` call with a dynamic import, we have to wrap the `<Admin />` component with React's `<Suspense />` component.

The `<Suspense />` component wraps lazy-loaded components and allows you to render intermediary (loading) content until the component has loaded.

```
import React, {Suspense} from "react";
```

Alternatively, you can skip this `{Suspense}` named import and use the `<React.Suspense />` component instead.

Then, we'll update the code to wrap `<Admin />` with `<Suspense />`:

```
import React, {Suspense} from "react";
import {createRoot} from "react-dom/client";
const Admin = React.lazy(() => import("./Admin.js"));

function User(props) {
    return (<>
        <p>User details show up here</p>
        {props.isAdmin && <Suspense>
            <Admin />
        </Suspense>}
    </>);
}
```

Notice how the `<Admin />` component is wrapped with the `<Suspense />` component.

Step 3: Provide intermediary/fallback content

The last step is where you have to define what content React will render **while** the component is loading (while the `import()` call is going to the network and fetching the file).

This is done by providing a `fallback` prop to the `<Suspense />` component. The `fallback` prop can take any JSX or Component. So, you change the `<Suspense><Admin /></Suspense>` into:

```
<Suspense fallback={<div>Loading admin ... </div>}>
```

If you had `<CircleSpinner />` component (an example of a loader), then you can also use it as a fallback as follows:

```
<Suspense fallback={<CircleSpinner />}>
    <Admin />
</Suspense>
```

Final result & comparison

Here's what the original code looks like (before lazy loading):

```
import React, {Suspense} from "react";
import {createRoot} from "react-dom/client";
import Admin from "./Admin.js";

function User(props) {
    return (<>
        <p>User details show up here</p>
        {props.isAdmin && <Admin />}
    </>);
}
```

And after lazy loading with React.lazy and Suspense:

```
import React, {Suspense} from "react";
import {createRoot} from "react-dom/client";
const Admin = React.lazy(() => import("./Admin.js"));

function User(props) {
    return (<>
        <p>User details show up here</p>
        {props.isAdmin && <Suspense fallback={<div>Loading
admin ... </div>}>
            <Admin />
```

```
}
```

When to lazy load?

Do not lazy load every single component. This will make your app slower. Generally, try to lazy load components that are very large and that do not show up immediately above the fold.
It's good practice to experiment with the result before and after and see how it affects the loading experience.

Recap

- To use React.lazy and suspense, follow the three steps below:
- Step 1: Static import to dynamic import with React.lazy
- Step 2: Wrap with Suspense
- Step 3: Provide intermediary/fallback content
- Do not lazy load every single component. This will make your app slower.
- Try to lazy load components that are very large and that do not show up immediately above the fold.

Was this helpful?

Introduction to testing

Skip chapter?

Last updated July 2022

This chapter contains lessons that explain the basics of testing a React component when using create-react-app.

When writing tests, you generally need two tools:

1. A testing library
2. A test runner

Testing library

A testing library is a library that allows you to assert (or verify) that a certain value or behavior matches the expectation. For example, assuming a function, `sum`, here's a pseudo-code for some code that tests the function:

```
assert(sum(2, 4)).to.be.equal(6);
assert(sum(1, 1)).to.be.equal(2);
```

All the functions in this code (except for `sum`) are provided by a *testing library*.

Once you have code to test your component, you need a library that runs those tests.

A test runner

A test runner is a tool that runs your testing code. If all the tests succeed, it tells you that the tests were successful. If one of the tests fails, it will notify you where and why it failed.

create-react-app

1. react-testing-library (the testing library).
2. jest (the test runner).

It's important to know these two tools/libraries so that you can read the appropriate documentation when necessary.

Furthermore, `create-react-app` allows you to run the tests with the `npm test` command. Run `npm test` in a fresh React app, and you should get a result similar to the below screenshot:

```
PASS  src/App.test.js
  √ renders learn react link (16 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.074 s
Ran all test suites.

Watch Usage: Press w to show more.
```

Note that you have to press the character a after running `npm test` to run all the tests.

This command will iterate over all your `src/` files looking for those that end with `.test.js`.

If you're following along, open the `src/App.test.js` file. We'll explain its code in the next lesson.

Recap

- When writing tests, you generally need two tools:
- 1. A testing library
- 2. A test runner

1. **react-testing-library** (the testing library).
2. **jest** (the test runner).

- You can run the tests with `npm test`. You need to press a to run all the tests.

Was this helpful?

Basic testing syntax

Skip chapter?

Last updated July 2022

When you open `src/App.test.js`, you should see a code very similar to the following:

```
import { render, screen } from '@testing-library/react';
import App from './App';

test('renders learn react link', () => {
  render(<App />);
  const linkElement = screen.getByText(/learn react/i);
  expect(linkElement).toBeInTheDocument();
});
```

Let's break it down step by step.

Goal

The goal of this test file is to test the `<App />` component. We're checking that it renders the text `learn react` anywhere in its content.

Code breakdown

We start by importing the component we'd like to test: `import App from './App';`.

Creating a test:

Then, we create our first test with the `test(description, callback)` function. This `test()` function is provided by `jest`.

```
test('renders learn react link', () => { ... })
```

or succeeds, you will see a ✅ or a ❌ next to this description. It's important to provide meaningful descriptions, especially once you start having dozens and hundreds of tests.

Rendering a component:

Then, inside the callback function of this test, we render the `<App />` component with the following code:

```
render(<App />);
```

It's also possible to render props. For example:

```
render(<App theme="dark" />)
```

The `render` method is provided by `react-testing-library`. So, we have to import it from `@testing-library/react`.

Looking for a string:

Then, we use the `screen.getByText()` function to look for the text `learn react` anywhere on the page where we rendered `<App />`.

`screen.getByText()` accepts a string as well as a regular expression. In this example, we pass the regular expression `/learn react/i` which searches for `learn react` with case-insensitive. So, it'll also match `Learn REACT`, for example.

The `screen` object is provided by `react-testing-library`. So, to import `render` and `screen`, you need to have the following import statement: `import { render, screen } from '@testing-library/react';`.

Asserting the result:

The last step consists of checking whether the text was indeed found in the rendered component.

This is where we `expect(linkElement).toBeInTheDocument();`.

provided by `react-testing-library`.

react-testing-library

Jest already includes some basic assertions. However, these are often not enough when testing a React application. The `react-testing-library` augments jest with new assertions such as `toBeInTheDocument`.

Was this helpful?

Your first test

Skip chapter?

Last updated July 2022

Let's say we've got the same `<Button />` component that we defined back in [Project I](#).

```
export default function Button(props) {
    return <button className="ui-button" type={props.type}
disabled={props.disabled}>{props.children}</button>;
}
```

Create a `Button.js` file in the `src/` folder. Then, create a `Button.test.js` file in that same folder. It's recommended to keep the test file in the same directory as the component you'd like to test. One of the benefits of that is that the `import` statements will be simplified. This will become evident later in this lesson.

There are many aspects that could be tested about this component. We'll focus on two:

1. Rendering `<Button />` should render a `button` element with a class `ui-button`.
2. Rendering `<Button disabled />` should render a `button` element with the `disabled` property set.

Test from the outside

While there is no state in this component, keep in mind that it's better to test the component from the outside. Your goal is not to test its internals. You don't need to know or care how the state or logic works inside the component. You should aim to test it from the outside. For example, I render a `<Button>Login</Button>` component. I expect to see a `<button className="ui-button">Login</button>`.

npm test

run the test file as you change it.

Button.test.js

Inside `Button.test.js`, we'll need to import the `Button` component as well as the `screen` and `render` methods:

```
import { render, screen } from "@testing-library/react";
import Button from "./Button.js";
```

First test

Next, we'll create the first test:

```
it("should render a button element with a 'ui-button' class",
() => {
    render(<Button />);

});
```

Once we render the `<Button />`, we need to find the `button` element. We can use the `screen.getByRole` method:

```
const buttonElement = screen.getByRole("button");
```

Then, we need to assert that this `buttonElement` was found with:

```
expect(buttonElement).toBeInTheDocument();
```

and check that the class `ui-button` is found in the element's classList.

Alternatively, you can check the `className` (string) property.

Putting it all together:

```
import { render, screen } from "@testing-library/react";
import Button from "./Button.js";

it("should render a button element with a 'ui-button' class", ()
=> {
  render(<Button />);
  const buttonElement = screen.getByRole("button");
  expect(buttonElement).toBeInTheDocument();
  expect(buttonElement.classList.contains("ui-
button")).toBe(true);
});
```

Second test

For the second test, we'll add another `it(" ... ", callback)`:

```
it("should have a disabled attribute when it receives a
disabled prop", () => {

});
```

Inside of it, we need to render the `Button` component with the `disabled` prop:

```
render(<Button disabled />);
```

Then, we'll find the `button` element the same way as we did before and then assert that it has the `disabled` property set to `true`:

```
const buttonElement = screen.getByRole("button");
expect(buttonElement.disabled).toBe(true);
```

Putting it all together, you get the following code in `Button.test.js`:

```
import { render, screen } from "@testing-library/react";
import Button from "./Button.js";

it("should render a button element with a 'ui-button' class", ()
⇒ {
  render(<Button />);
  const buttonElement = screen.getByRole("button");
  expect(buttonElement).toBeInTheDocument();
  expect(buttonElement.classList.contains("ui-
button")).toBe(true);
});

it("should have a disabled attribute when it receives a disabled
prop", () ⇒ {
  render(<Button disabled />);
  const buttonElement = screen.getByRole("button");
  expect(buttonElement.disabled).toBe(true);
});
```

It takes a while to get used to all the possible assertions. However, remember that it's not about memorizing but rather knowing where to search for them.

Was this helpful?

Testing user events

Skip chapter?

Last updated July 2022

In this lesson, you'll learn how to write tests for components that rely on user events such as `onClick={ ... }`.

Let's start by creating a `Counter.js` file in the `src/` folder based on [Chapter 15's Counter](#) challenge.

```
import {useState} from "react";
import {createRoot} from "react-dom/client";

export default function Counter() {
    const [times, setTimes] = useState(0);

    return (<>
        <h2>{times} times clicked</h2>
        <button onClick={() => setTimes(times + 1)}>Add
1</button>
    </>);
}
```

We'll write the necessary code to test the following functionality:

1. When the component renders, we see `0 times clicked`.
2. After clicking on the button the first time, we see `1 times clicked`.
3. After clicking on the button another time, we see `2 times clicked`.

This is exactly how React Tutorial tested your implementation of the `Counter` component.

Counter.test.js

as well as the `screen` and `render` methods. We'll need to click the button in this test programmatically, so, we'll also need the `fireEvent` from `react-testing-library`:

```
import { render, screen, fireEvent } from "@testing-library/react";
import Counter from "./Counter.js";
```

Let's create the test and render the `<Counter />`:

```
it("should increment the counter on every click", () => {
  render(<Counter />);

});
```

Then, we'll start by checking that we can find `0 times clicked` before clicking the button:

```
const text0 = screen.getByText(/0 times clicked/i);
expect(text0).toBeInTheDocument();
```

Afterward, we'll find the button and click it with `fireEvent.click(element)`:

```
const button = screen.getByRole("button");

// click Add 1
fireEvent.click(button);
```

After clicking the button, we should expect to find `1 times clicked`:

```
const button = screen.getByRole("button");

// click Add 1
fireEvent.click(button);
```

We can also repeat the same logic again and check for `2 times clicked`:

```
// click Add 1 (another time)
fireEvent.click(button);
const text2 = screen.getByText(/2 times clicked/i);
expect(text2).toBeInTheDocument();
```

Putting it all together, we end up with:

```
import { render, screen, fireEvent } from "@testing-
library/react";
import Counter from "./Counter.js";

it("should increment the counter on every click", () => {
  render(<Counter />);

  // 0 times clicked before clicking
  const text0 = screen.getByText(/0 times clicked/i);
  expect(text0).toBeInTheDocument();

  const button = screen.getByRole("button");

  // click Add 1
  fireEvent.click(button);
  const text1 = screen.getByText(/1 times clicked/i);
  expect(text1).toBeInTheDocument();

  // click Add 1 (another time)
  fireEvent.click(button);
  const text2 = screen.getByText(/2 times clicked/i);
  expect(text2).toBeInTheDocument();
});
```

Notice how we did not need to care about the internals of this component. It doesn't matter what the state variable inside of this component is called. We only test its

Was this helpful?

Intro to redux

Skip chapter?

Last updated December 2022

The goal of this optional chapter is to introduce **redux**.

You can read below to see when to use Redux and when not to use it. If you decide to skip this chapter, you should also skip the following chapters as they are all interconnected.

What is redux

We will leave the exact definition of redux for the next lesson. For now, `redux` is a library that you can use to manage **state** in large and complicated applications.

When to use redux

If you have a small application, redux **will be overkill**. There's no way around that. The code will look a lot more complicated than it should. However, as your application grows in terms of components and developers working on the project, the benefits of redux start shining.

So, it's safe to assume that in a lot of cases you won't be needing redux. Do not use it because it's cool or trendy. In general, here are some reasons that can make you consider using redux:

- Large number of developers on the project.
- You have a lot of `state` variables that are being used in multiple components.
- You are already very familiar with redux and using it will be considered a boost in productivity as opposed to a hindrance.
- Your app has a lot (and/or very frequent) state updates (referring to state that is used in multiple components).

Lifting state up can quickly become cumbersome as the number of state variables that you need to lift up grows. Also, sometimes you need to lift state several components high (more than 5 components up) and that also becomes cumbersome.

You won't have this problem with `redux` as every component will select the particular data it needs as well as update the state.

Redux vs Context

We learned back in the Context chapters that when a state variable is updated in a Context, all the components subscribed to that context will need to update. This is totally fine for low-frequency updates such as toggling a theme or changing the app language.

However, more frequent global state updates would cause performance issues. In that case, redux would make more sense here. Also, with a large number of state being updated, redux makes it easier to keep track of things.

So, generally, the smaller/easier an app is to build, the less you need to use redux.

Was this helpful?

What is redux

Skip chapter?

Last updated March 2024

Redux is a library that allows you to predictably update your app state.

For the sake of completeness, here's the official definition of redux from the [Redux documentation](#):

> Redux is a pattern and library for managing and updating application state, using events called "actions". It serves as a centralized store for state that needs to be used across your entire application, with rules ensuring that the state can only be updated in a predictable fashion.

Don't worry too much about it. The definition contains some terms that we have not defined yet (such as `store` and `actions`).

Import Cost

The redux library is pretty small. The redux JavaScript library weighs around 4kb. However, as we'll see in the following chapters, the create-react-app's redux template uses additional packages so that will drive up the import cost higher. We'll talk about this in the following chapters.

Why is redux 'complicated' (at first)

You may have tried to learn redux at first and thought it's complicated. You're not alone. In this course, we'll break down the concepts one by one so that it's not overwhelming. The challenges are broken up into smaller pieces so that you understand every topic and see at the end how it all comes together. So, don't worry about it!

There are 4 core concepts that you need to understand. Once you understand these 4 core concepts, it'll become easier for you to work with redux. In the next lesson, we'll

Recap

- Redux is a library that allows you to predictably update your app state.
- The redux JavaScript library weighs around 4kb.
- There are four core concepts that you need to understand to be able to work with redux.

Was this helpful?

State, Store, and Reducer

Skip chapter?

Last updated March 2024

In this lesson, we'll break down 3 of the four core concepts of redux:

1. State

Just like in React, `state` in redux refers to a variable that may be updated later on. Changes to this state object will have consequences on the UI.
While in React, we could create a state that stored a boolean value, a number, or a string, etc., in Redux, the state will always be an object.

Throughout this chapter, we'll build a counter app using redux (there won't be any React code in this and the following chapter). This is usually the first thing that you learn to build with redux. It may sound boring but it's actually a great way to learn Redux.

Note: since a counter app is not complicated, building it with redux will be more complicated than necessary. But this is fine for now, as we need to understand the core concepts.

For our counter application, the following state object would make sense:

```
const state = {
    value: 0 // starting value of the counter
};
```

The name of the variable doesn't matter much. What's important is that the `state` is an object. And, in this case, has a key `value` with an initial value of 0.

As with React, we are not supposed to modify this object directly. Instead, we'll use a reducer function to do so.

A reducer is a function that accepts two parameters: the current state and an action. Depending on that action, it may return a new state. Otherwise, it will return the current state as is.

For example, let's take a look at this *incomplete* example:

```
const initialState = {
    value: 0,
};

const counterReducer = (state = initialState, action = {}) => {
  return state; // return the state as is
};
```

This is the most basic reducer. It receives the current state and returns it exactly the way it is. Notice the parameter `state = initialState`. This means that if we call counterReducer without any parameter, it will default to the initialState. This will happen the first time we call this reducer.
Using the code above, here's how we can use this reducer:

```
const state = counterReducer(); // state will default to
initialState
console.log(state); // {value: 0}
```

Now, let's make it possible to **increment** the counter. For that, we'll pass an action of `{type: "counter/increment"}`. An action is an object that is used to make a change to the state.

So, we need to update the reducer to handle this action with an `if (action.type === "counter/increment")`:

```
const initialState = {
    value: 0,
};
```

```
    return {
        value: state.value + 1 // important: do NOT mutate the
state.
    };
  }

  return state; // return the state as is (in all other cases)
};
```

It's important to **not** mutate the state. Just like in React because redux also uses triple equality check to know whether the state has been updated or not.

The `state` argument contains the current state object of the application.

Notice how we check the `action.type` and when it's equal to `"counter/increment"`, we return a new state object that has the value immutably incremented.

Here's how this reducer can be used:

```
const state = counterReducer(); // state will default to
initialState
console.log(state); // {value: 0}
const newState = counterReducer(state, {type:
"counter/increment"});
console.log(newState); // {value: 1}
```

A couple of things to note here:

- The `action` is an object with a key `type`. This is by design to allow us to also send a payload. More information on this later in this chapter. For now, make sure to pass an object with the key `type`.
- You will never call the reducer function yourself. The above is only meant to show you how the state gets updated when the reducer gets called.
- You may have seen reducers implemented with `switch` cases instead of `if`. You can use whatever you prefer. If you've never seen that, you can disregard this

In redux, the reducer is a function that receives the current state and an action. Based on that action, it will return a new state (if an update is necessary). Otherwise, it will return the same state.

The word "reducer" has been chosen because redux reduces the actions in your application (over time) into a single state. This is a bit similar to how you can reduce an array of items into a single value.

Note: the above will only become clear once you go through the following challenges (which include a visualization challenge) that we've prepared for you. So, don't worry about not understanding everything.

3. Store

A *Store* is an object that contains the reducer and the state. Later on, when we want to make updates to the state, we will have to ask the store to make an update (we'll dispatch an action).
The store will then use the reducer method to compute the new update.

Assuming the `counterReducer` function from above, here's how you create a store in redux:

```
import { createStore } from "redux";

const store = createStore(counterReducer);
```

Putting it all together

Combining everything we learned in this lesson, we get:

```
import { createStore } from "redux";

const initialState = {
```

```
const counterReducer = (state = initialState, action = {}) => {
  if (action.type === "counter/increment") {
    return {
        value: state.value + 1 // important: do NOT mutate the
state.
    };
  }

  return state; // return the state as is (in all other cases)
};

const store = createStore(counterReducer);
```

Time to go through the following challenges that will make all the above clearer!

Recap

- We'll build a counter app to learn how redux works. Considering that a counter app is usually a relatively simple task (the app contains only 1 state value), redux will seem like overkill. This is fine for learning purposes.
- 1. State: In redux, the state is an object that will most likely be updated later on. Changes to this state object will have consequences on the UI.
- 2. A reducer is a function that accepts two parameters: the current state and an action. Depending on that action, it may return a new state. Otherwise, it will return the current state as is.
- 3. A *Store* is an object that contains the reducer and the state. We will, later on, use this store to request updates to the state.

Was this helpful?

Dispatching actions

Skip chapter?

Last updated May 2022

The fourth core concept of redux is dispatching actions.

We are not allowed to manually make changes to the state stored in the store. Instead, we have to ask the store to do the change for us.

We do that by dispatching (sending) an action.

We've seen that actions are objects with the following structure:

```
{
    type: "counter/increment", // example of an action
}
```

In a later chapter, we'll see how you can also provide a `payload`. For example, increment by 5 or increment by 10.

To dispatch an action, you have to use the `store.dispatch(action)` function:

```
store.dispatch({type: "counter/increment"});
// another example
store.dispatch({type: "counter/reset"});
```

Calling the `store.dispatch()` method will make the store call the reducer to compute the new state. If the state has changed, your UI will be triggered so it can update (this is covered in the react-redux chapter).

Dispatching on click

You can dispatch an action anywhere in your code. A common use case is dispatching when a button is clicked. This is what we've done in the counter app:

```
addButton.addEventListener("click", () => {
    store.dispatch({ type: "counter/increment" });
});
```

Putting it all together

```
import { createStore } from "redux";

const initialState = { value: 0 };

const counterReducer = (state = initialState, action) => {
    if (action.type === "counter/increment") {
        return {
            value: state.value + 1 // important: do NOT mutate
the state.
        };
    }

    return state; // return the state as is (in all other cases)
};

const store = createStore(counterReducer);

const addButton = document.querySelector("#add-button");

addButton.addEventListener("click", () => {
    store.dispatch({ type: "counter/increment" });
});
```

In summary

How redux works can be summarized with the below description:

When you need to make changes to the UI, you have to dispatch an action on the store.

The store will compute the new state using the reducer function and the action received.

Recap

- We are not allowed to manually make changes to the state stored in the store. Instead, we have to ask the store to do the change for us. We do that by dispatching (sending) an action.
- To dispatch an action, you have to use the `store.dispatch(action)` function.
- Calling the `store.dispatch()` method will make the store call the reducer to compute the new state. If the state has changed, your UI will be triggered so it can update (this is covered in the react-redux chapter).
- Redux can be summarized with the below description:
  - The store contains the application state and a reducer function.
  - When you need to make changes to the UI, you have to dispatch an action on the store.
  - The store will compute the new state using the reducer function and the action received.

Was this helpful?

Chapter recap

Skip chapter?

Last updated February 2022

Nice work!

In this chapter, you learned about the core concepts of redux.

In the following chapter, we'll learn about the Redux Toolkit (`@reduxjs/toolkit`) which is the redux library used in the `create-react-app`'s redux boilerplate (that we'll talk about in the react-redux chapter).

So, after this chapter, you will never `import` from `"redux"` anymore. This is why we did not have you practice the `import {createStore} from "redux"`.

Also, we skipped the `store.subscribe()` method because it'll be completely different once we work with redux inside a React app!

Redux dev tools

If you're enjoying redux and think might use it in the future, it may be helpful to install a **Redux dev tools extension** for your browser.

Since your state is updated predictably (and immutably) with dispatched actions, the extension can offer a neat feature called Time Travel that allows you to visualize your application state at every point in time (after every dispatched action)!

Chapter Recap

- Redux is a library that allows you to predictably update your app state.
- The redux JavaScript library weighs around 4kb.
- There are 4 core concepts that you need to understand to be able to work with redux.
- We'll build a counter app to learn how redux works. Considering that a counter app is usually a relatively simple task (the app contains only 1 state

later on. Changes to this state object will have consequences on the UI.

- 2. A reducer is a function that accepts 2 parameters: the current state and an action. Depending on that action, it may return a new state. Otherwise, it will return the current state as is.
- 3. The store is an object that contains the reducer and the state. We will, later on, use this store to request updates to the state.
- We are not allowed to manually make changes to the state stored in the store. Instead, we have to ask the store to do the change for us. We do that by dispatching (sending) an action.
- To dispatch an action, you have to use the `store.dispatch(action)` function.
- Calling the `store.dispatch()` method will make the store call the reducer to compute the new state. If the state has changed, your UI will be triggered so it can update (this is covered in the react-redux chapter).
- Redux can be summarized with the below description:
  - The store contains the application state and a reducer function.
  - When you need to make changes to the UI, you have to dispatch an action on the store.
  - The store will compute the new state using the reducer function and the action received.

Was this helpful?

Intro to Redux Toolkit

Skip chapter?

Last updated September 2022

When you use `create-react-app` and use the official *redux* template (we will see that in the next chapter), you will notice that it uses the Redux Toolkit.

The **Redux Toolkit** is the official, opinionated, batteries-included way of writing redux.

The Redux Toolkit was created to make common redux tasks easier and more efficient to write.

We'll explore the Redux toolkit in this chapter by rewriting the same examples and challenges from the last chapter. This will serve as a preparation for the next chapter: **Intro to react-redux**.

Package name

While the name of the redux package is `redux`, the name of the Redux Toolkit package is `@reduxjs/toolkit`.

Import cost

The size of the `@reduxjs/toolkit` package is approximately 32kb minified.
Note that this number includes the 4kb that's coming from the `redux` library.
The reason for this jump in size is that the `@reduxjs/toolkit` package includes the **immer** library which we'll introduce below and explain throughout this chapter.

Immutability with immer

`immer` is a JavaScript library that allows you to *immutably* update the state while still writing the same JavaScript code you're used to.

such as objects, arrays, and arrays of objects.

We'll see in this chapter how we can simplify the reducer code because `immer` is loaded by `@reduxjs/toolkit`.

Recap

- When you use `create-react-app` and use the official *redux* template, you will notice that it uses the Redux Toolkit.
- The Redux Toolkit was created to make common redux tasks easier and more efficient to write.
- While the name of the redux package is `redux`, the name of the Redux Toolkit package is `@reduxjs/toolkit`.
  The size of the `@reduxjs/toolkit` package is approximately 32kb minified.
- `immer` is a JavaScript library that allows you to *immutably* update the state while still writing the same JavaScript code you're used to.

Was this helpful?

Slices

Skip chapter?

Last updated January 2022

Configuring a new store tends to be a bit complicated, especially when the store grows to accommodate more than one *category* of actions. For example, previously, we had had actions related to the `counter`. It will get more complicated if you had to handle additional actions for `app` for example (where the user can customize the app settings).

This is why the Redux Toolkit introduces the concept of *slice*.

A Redux Slice is a collection of reducer logic and actions for a single feature of our app.

For example, we'd have a `counter` slice and then an `app` slice. These two slices would be configured separately but provided to the same store.

Let's see how we can create a slice and configure the store in Redux Toolkit:

```
import { createSlice } from "@reduxjs/toolkit";

const counterSlice = createSlice({
  name: "counter",
  initialState: {
    value: 0,
  },
  reducers: {
    increment: (state) => {
      // TODO: write reducer code here
    }
  },
});
```

Notice how we import the `createSlice` function from `@reduxjs/toolkit` and then call this function and provide the `name`, `initialState`, and `reducers`.

(remember the `counter/increment`, `counter/decrement`, etc.). That's because this slice is only concerned with actions related to the `counter` that we provided in the `name`.

Writing the reducer (and benefiting from immer)

In the code above, we created one reducer function called `increment`. In this reducer function, we are allowed to write code that *looks like* it's mutating the state. This is because *immer* is being loaded by Redux Toolkit. So, the code that looks like its mutating the state is actually running in an immutable way.

Immer creates a draft state from your changes and then produces a brand new immutable state based on those changes.

So, our `increment` reducer code would look like this:

```
// ...
 reducers: {
   increment: (state) => {
     // While this looks like it's mutating the state, immer
creates a draft state and then produces immutable state changes
     state.value += 1;
   }
 },
 // ...
```

If you had an example with an array of objects, `immer` would make it much easier for you to update that state immutably.

Note that *it just works*. You may be wondering if you should import `immer` or call something. But you don't have to do anything. `immer` is already loaded by `@reduxjs/toolkit` and it wraps all the reducer functions.

Configuring the store

```
import { configureStore, createSlice } from "@reduxjs/toolkit";

const counterSlice = createSlice({
  name: "counter",
  initialState: {
    value: 0,
  },
  reducers: {
    increment: (state) => {
      // While this looks like it's mutating the state, immer
creates a draft state and then produces immutable state changes
      state.value += 1;
    },
  },
});

const store = configureStore({
  reducer: counterSlice.reducer,
});
```

Notice how we pass `counterSlice.reducer` (not `reducers`). This is different than the `reducers` that you defined in the `createSlice()` function.

The `counterSlice.reducer` is created automatically by the Redux Toolkit.

Recap

- A Redux Slice is a collection of reducer logic and actions for a single feature of our app.
- Immer creates a draft state from your changes and then produces a brand new immutable state based on those changes.

Was this helpful?

Actions & payloads

Skip chapter?

Last updated March 2022

In this lesson, we'll take a look at actions with Redux Toolkit as well as supplying payloads with an action.

Actions

Back with `redux` we dispatched an action using the following code:

```
store.dispatch({type: "counter/increment"});
```

With Redux Toolkit we get a function that we can extract from `slice.actions`. We can then use this function in conjunction with `store.dispatch()`. Let's see how.

We'll start off with the same code as the previous lesson and extend it with the *dispatch* code:

```
import { configureStore, createSlice } from "@reduxjs/toolkit";

const counterSlice = createSlice({
  name: "counter",
  initialState: {
    value: 0,
  },
  reducers: {
    increment: (state) => {
      // While this looks like it's mutating the state, immer
creates a draft state and then produces immutable state changes
      state.value += 1;
    },
  },
});

const store = configureStore({
```

```
// extract the increment function from counterSlice.actions
const { increment } = counterSlice.actions;

document.querySelector("#add").addEventListener("click", () => {
  // dispatch an increment() action
  store.dispatch(increment());
});
```

The `increment` function is available under `counterSlice.actions`. Its name matches the name of the function we provided under the `reducers` above.

You can either use object destructuring as we did in the example above or access the function with `counterSlice.actions.increment`.

If you have more functions (when you have more reducers in your slice), you can extract them in the same way:

```
const { increment, decrement } = counterSlice.actions;
```

Supplying payloads

We've seen so far how to work with actions that do not take any payload. They always perform the same action (for example, increment by 1 or decrement by 1).

Sometimes, however, you'd like to make an action that increments by a number (sometimes 1, sometimes 5, and sometimes another number). This value is called a *payload*.

Every reducer receives a 2nd argument which is called the `action`. You can access the `action.payload` to get access to value that was passed to the action when it got called. Let's see an example by implementing an `incrementBy(value)` feature:

```
import { configureStore, createSlice } from "@reduxjs/toolkit";

const counterSlice = createSlice({
```

```
    value: 0,
  },
  reducers: {
    incrementBy: (state, action) => {
        console.log(action.payload); // visualize payload
        state.value += action.payload;
    }
  },
});

const store = configureStore({
  reducer: counterSlice.reducer,
});

const { incrementBy } = counterSlice.actions;

document.querySelector("#add").addEventListener("click", () => {
  // the payload is 1
  store.dispatch(incrementBy(1));
});

document.querySelector("#add-five").addEventListener("click", ()
=> {
  // the payload is 5
  store.dispatch(incrementBy(5));
});
```

Notice how we can now call `incrementBy(1)`, `incrementBy(5)`, or any other value here. This value will be received as the `action.payload` which you can then use to update the state.

Recap

- With Redux Toolkit we get a function that we can extract from `slice.actions`. We can then use this function in conjunction with `store.dispatch()`.

access the action.payload to get access to value that was passed to the action when it got called.

Was this helpful?

Intro to React Redux

Skip chapter?

Last updated April 2024

Now that we've learned about Redux and the Redux Toolkit, we're ready to learn about using Redux in a React app.

You can generate a new React app with the *redux* template by running the following command. However, this will opt into using TypeScript too.

```
npx degit reduxjs/redux-templates/packages/vite-template-redux
my-app
```

If you don't want to use TypeScript, you will have to use `create-react-app`, which is not maintained anymore (so that's not recommended):

```
npx create-react-app my-app --template redux
```

The react-redux package

In a react app, you will primarily work with the `@reduxjs/toolkit` that we learned in the previous chapter and a new package called `react-redux`.

The `react-redux` provides specific Components and Hooks that allow you to use Redux in a React application.

It weighs approximately 16KB minified.

Topics covered

In this chapter, we'll cover the following topics:

- Using Redux in a React app.
- Using the `Provider` and wrapping our app with this provider.

Using the `useDispatch` hook to dispatch actions.

The documentation of the React-Redux specifics is hosted at **react-redux.js.org**.

Redux vs local state

Back when we **refactored SuperM to use Context**, we have not left *all* the states in the context. We made some distinction between local state (that stayed in a component) and state that was used by multiple components and had to go in the context.

We'll apply the same logic when using Redux. This will be further explained in the next chapter as we refactor the final project into Redux.

Recap

- The `react-redux` package costs approximately 16KB.

Was this helpful?

Provider

Skip chapter?

Last updated August 2023

In a React app, you can make the `store` available to any child component by wrapping your entire app with the `Provider` component. This will allow you to dispatch events from any component inside the app (as long as it's a child of `<Provider />`) as well as use *selectors* (that we'll explain in this chapter).

For that, you will need to import the `Provider` from `react-redux` (as a named import) and then wrap your entire app with it. For example, let's say you had the following code:

```
import { createRoot } from 'react-dom/client';
import { store } from './store.js';

createRoot(document.querySelector('#react-root')).render(<App
/>);
```

In this example, we're assuming that we're importing a `store` (create with `@reduxjs/toolkit`'s `configureStore` method).

Now, we'll import the `Provider` and wrap our entire app with it and pass the `store` as a `store` prop:

```
import { createRoot } from 'react-dom/client';
import { store } from './store.js';
import { Provider } from 'react-redux';

const root = document.querySelector('#react-root');

createRoot(root).render(<Provider store={store}>
    <App />
</Provider>);
```

it accessible for all the children of the `Provider` component.

Store definition

The store definition that we assumed above is the same as it was in the previous chapter.

This may go in the same file, or it could also be defined in another file, exported, and then imported in the file where you need to use the `Provider`.

Recap

- In a React app, you can make the `store` available to any child component by wrapping your entire app with the `Provider` component.
- It's generally recommended that you wrap your entire app with the `Provider`. This will allow all child components to access the `store` (as well as dispatch actions on it).
- The `store` can be defined in the same file or imported from another file.

Was this helpful?

The useSelector hook

Skip chapter?

Last updated February 2022

The `useSelector` hook allows you to extract data from your redux store.

It accepts a callback function that receives the entire state as an argument. You can then decide which data you'd like to get. Let's take a look at an example that assumes a store created with the following slice:

```
const counterSlice = createSlice({
  name: "counter",
  initialState: {
    value: 0,
  },
  reducers: {
    increment: (state) => {
      state.value += 1;
    },
  },
});
```

A `<Counter />` component will need to display the counter value. Look at the `initialState`. It's an object with the *key* `value` that we need to extract.

```
import {useSelector} from "react-redux";

function Counter() {
    const counter = useSelector(state => state.value);

    return <h1>Counter: {counter}</h1>
}
```

The code above imports the `useSelector` hook from the `react-redux` package. It then uses this hook and passes the following callback function:

This is an arrow function that extracts the counter value from the state.

Subscribe to changes

When you use the `useSelector` hook inside of a component, this component will subscribe to the changes of that store. This means that whenever the state of that store changes, the component will be called again, triggering the `useSelector` hook again.

Recap

- The `useSelector` hook allows you to extract data from your redux store.
- The `useSelector` hook accepts a callback function that receives the entire state as an argument.

Was this helpful?

The useDispatch hook

Skip chapter?

Last updated July 2022

The `useDispatch` hook returns a reference to the `dispatch` function from the Redux store. This allows you to call `dispatch()` with an action inside of it like we did in the previous chapter.

Let's take a look at a code example that assumes the following `counter` store and `increment` action defined in `store.js`:

```
import { configureStore, createSlice } from "@reduxjs/toolkit";

const counterSlice = createSlice({
  name: "counter",
  initialState: {
    value: 0,
  },
  reducers: {
    increment: (state) => {
      state.value += 1;
    },
  },
});

const store = configureStore({
  reducer: counterSlice.reducer,
});

const { increment } = counterSlice.actions;

export {store, increment};
```

Here's how we can complete our `Counter` component to increment the counter on click:

```
export default function Counter() {
    const dispatch = useDispatch();

    return <button onClick={() => dispatch(increment())}>Add
1</button>;
}
```

We started by importing the `useDispatch` hook from `react-redux`.
Then, we used that hook to extract the `dispatch` function: `const dispatch = useDispatch()`. More on this below.
We also imported the `increment` function (action) from the store.

Finally, when the user clicks on the button, we call `dispatch(increment())` just as we did in the previous chapter.

Remember the rules of hooks

A common mistake is trying to call `useDispatch` in the `onClick` handler. This will **NOT** work because of the rules of hooks.

You cannot call hooks inside a condition or an event handler. They should always be called at the top of the component. Feel free to refer back to the **rules of hooks lesson**.

Remember that the `useDispatch` hook does **not** allow you to dispatch an action. What it does, however, is return the `dispatch` function that you can call later when needed. This concept is a bit similar to the **Custom useFetch hook** where we had to call the `useFetch` hook to receive the `get` function back.

Importing the actions

One more note. Don't forget that you need to import the *actions* from where they were defined. In the example above, the `increment` action was defined in `store.js`, so we had to import it:

Recap

- The `useDispatch` hook returns a reference to the `dispatch` function from the Redux store. This allows you to call `dispatch()` with an action inside of it, as we did in the previous chapter.
- Remember the rules of hooks.
- Remember that the `useDispatch` hook does **not** allow you to dispatch an action. However, it does return the `dispatch` function that you can call later when needed.

Was this helpful?

Refactoring SuperM to Redux

Skip chapter?

Last updated December 2022

In this final redux chapter, we'll refactor SuperM to Redux using the knowledge we acquired in the last three chapters.

Chapter overview

To refactor SuperM into redux, we'll start by creating the store. The store will have two actions:

- `addProduct`
- `removeProduct`

Since the Redux Toolkit automatically includes *immer*, the implementation of these two actions will be easier than before. We'll be able to write code that looks like it's mutating the state while *immer* will be immutably performing the operation.

After creating the store, we'll create some selectors that we find in several places in the app, and then finally, we'll update our components to use these two actions and selectors.

As with the Context chapter, any local state can (and should) stay within the component. We only move the shared state to the redux store.

Was this helpful?