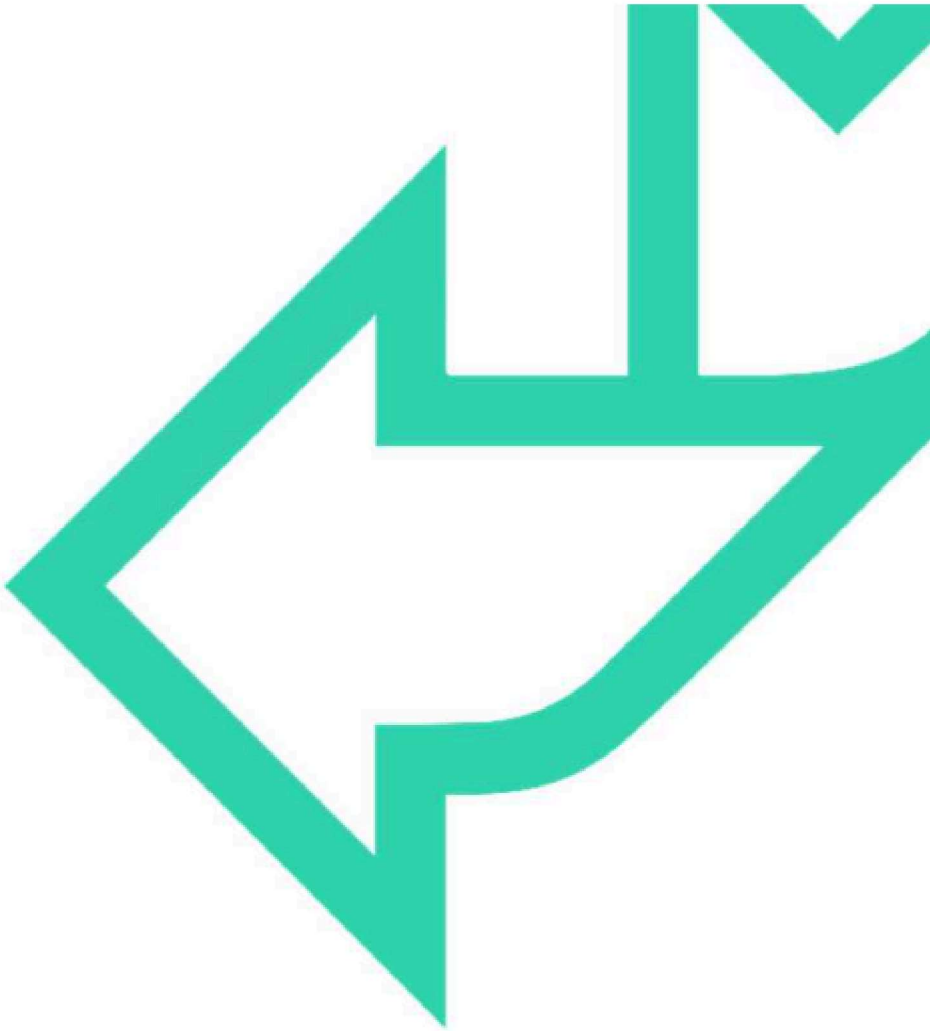
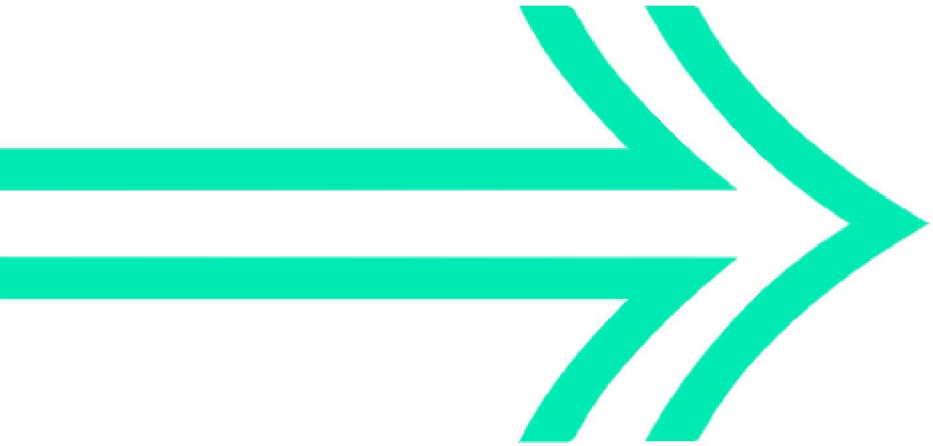
QA
POWERING POTENTIAL

BUILDING WEB APPLICATIONS WITH REACT



Introduction to React

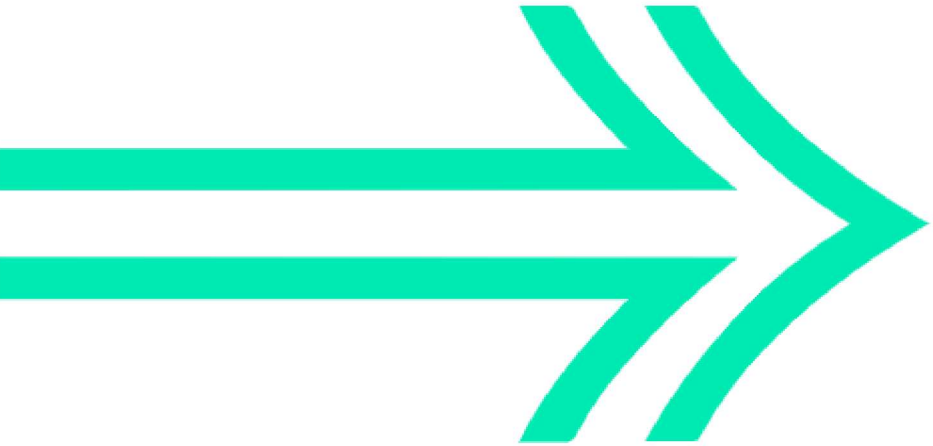
OBJECTIVES



In this module, you will:

- strengthen your understanding of **HTML, CSS, and JavaScript** fundamentals.
- develop the ability to write React code.
- apply problem-solving skills to **build a single-page application** using React.

GETTING THE MOST FROM THIS MODULE

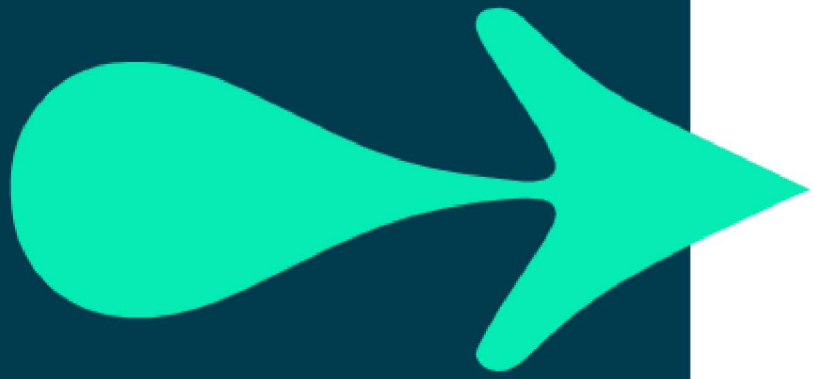


You should:

- be prepared to get hands-on.
- not expect to get everything right the first time.
- bring your problem-solving skills.
- ask and answer any questions you'd like to.
- take breaks – they're important!

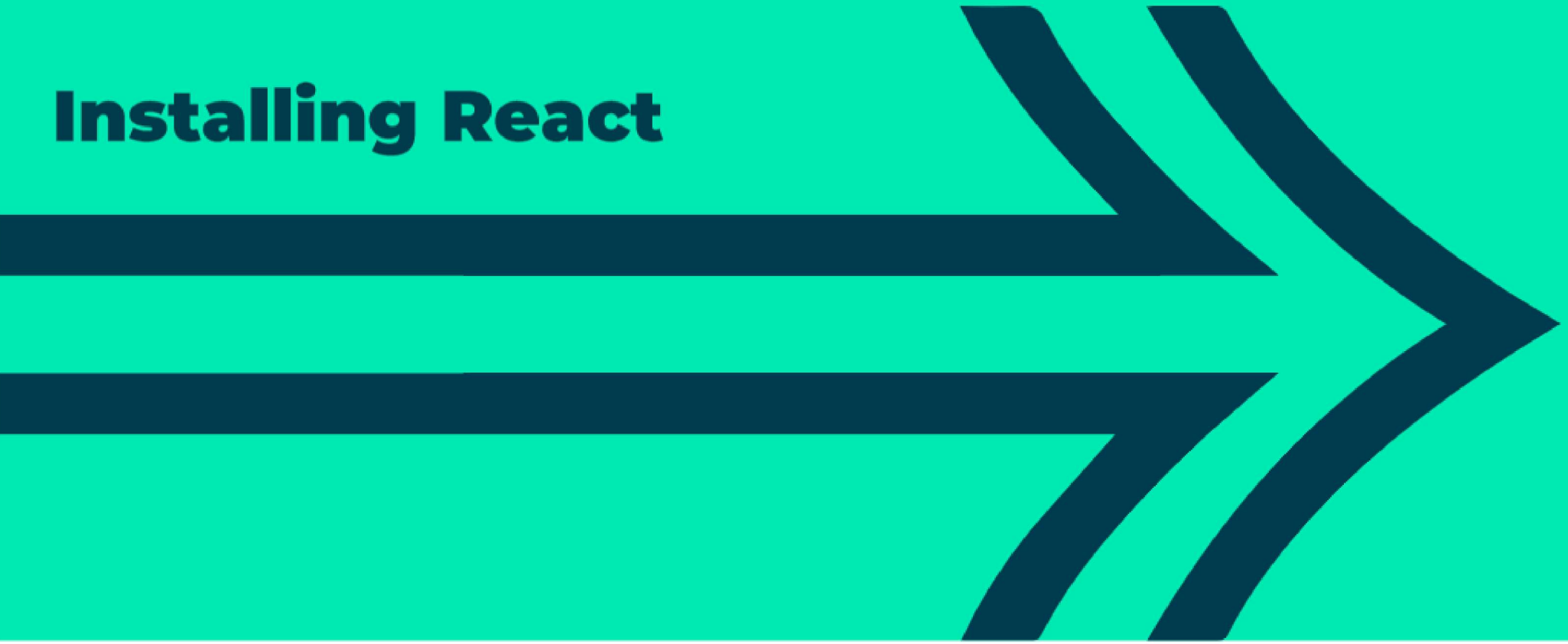
MODULE OVERVIEW

- Strengthen your web skills (with exercises)
- React overview
- Explore React concepts (with exercises)
- Hands-on project – build a web app yourself



Throughout the module, you'll be given time to progress through the hands-on project at your own pace.

Installing React



Built on Node

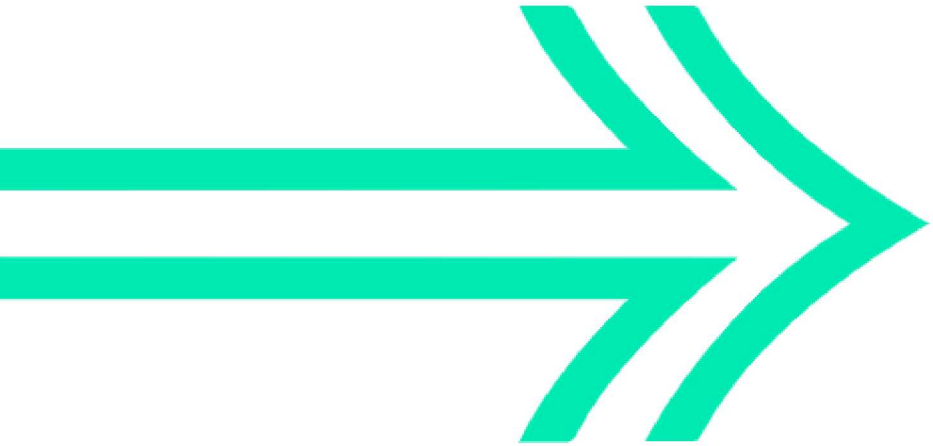
ReactJS is predictably built using node, although as you will see, there are many ways we can extend React from the base install.

Check that you have both node and npm installed. In a terminal in VSCode, type:

node --version

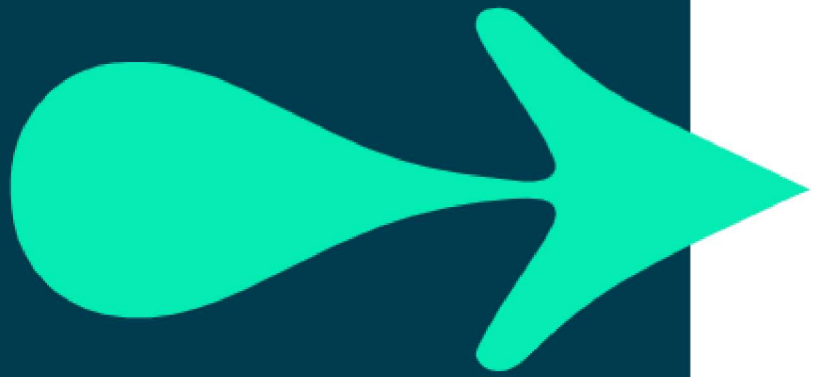
And then

npm --version



```
PS C:\Users\Uswer\Desktop\BAEREact files> node --version
v18.14.1
PS C:\Users\Uswer\Desktop\BAEREact files> npm --version
9.5.0
```

INSTALLING REACT



React is very easy to install in VSCode, just a few short commands and you have an install that runs exactly as you would hope.

The command below allows us to create an app in a folder base:

npx create-React-app appname

```
npx create-react-app solution
```

The process may take a couple of minutes. There will be some standard warnings (these are being worked on by the creators – ignore them!)

cd into the folder

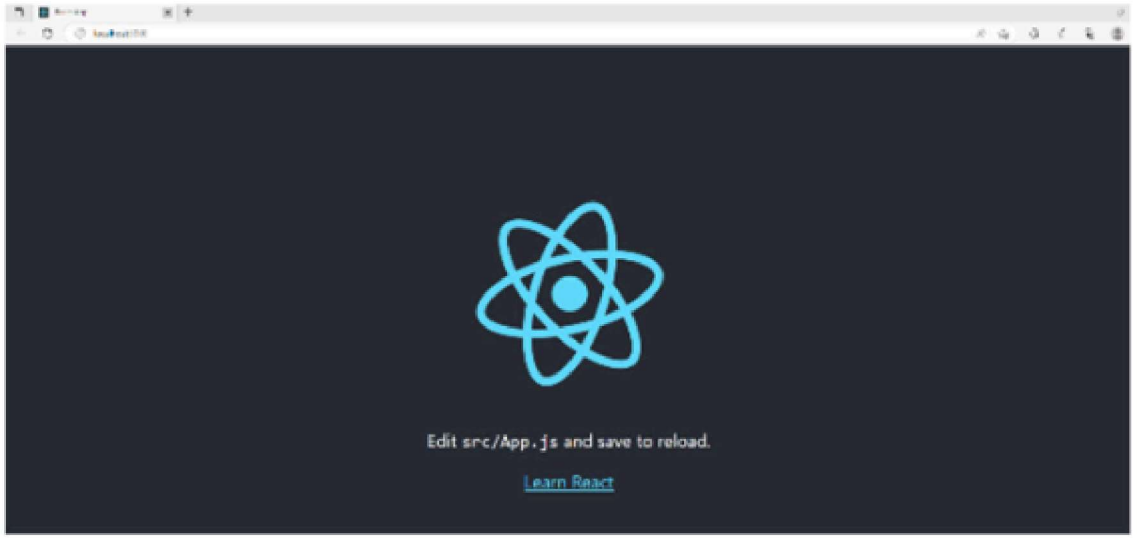
```
cd solution
```

npm start

```
npm start
```

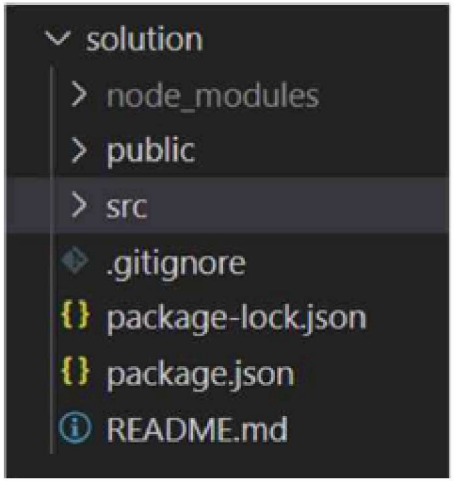
INSTALLING REACT

Gives you a live locally hosted webpage.

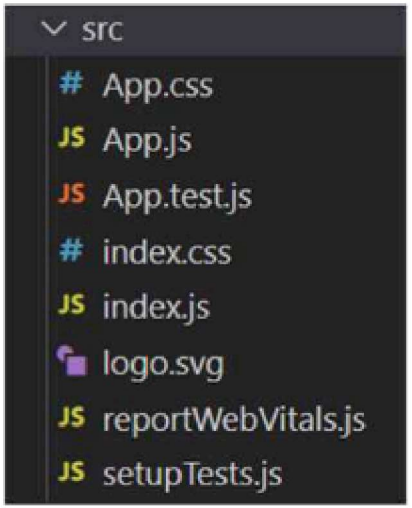


You are now ready to explore the framework.

STANDARD FILES



At first, the number of files is overwhelming, but we only use a small number of these to any extent.

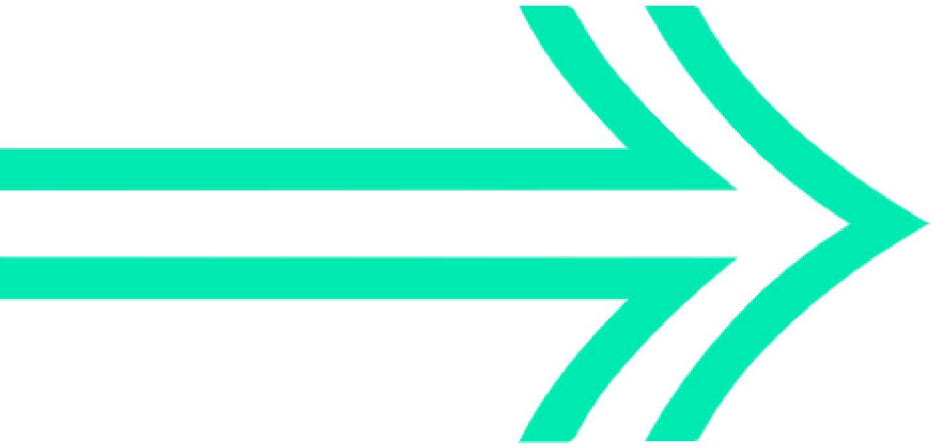


The files we edit to change our creation are in the src folder, so let's focus there.

Src
App.js

This is the file that is passed to the index.html page in the public folder for rendering.

All of the components we create will eventually be passed here to be displayed on screen.



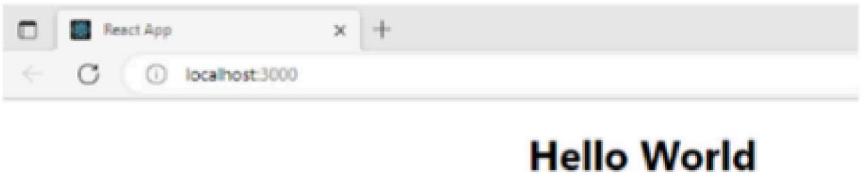
```
import logo from './logo.svg';
import './App.css';

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

export default App;
```

src
App.js(x)

Changing the file to this:

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

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

export default App;
```

Gives us

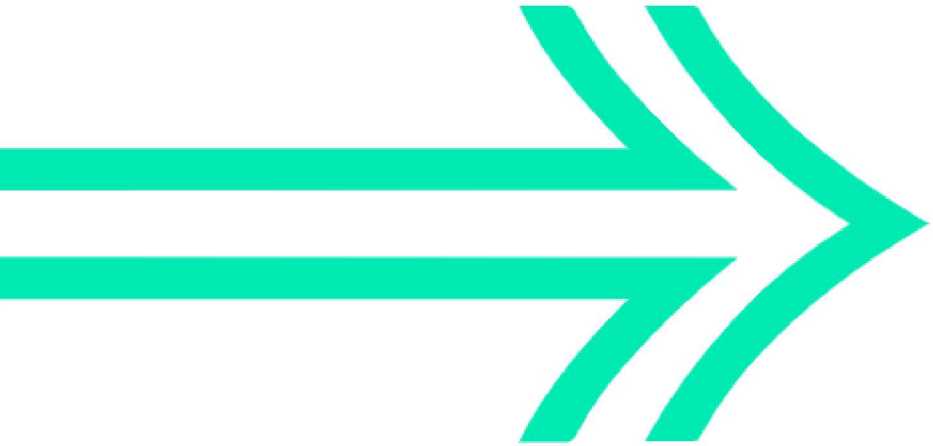


src
index.js
index.html

index.html

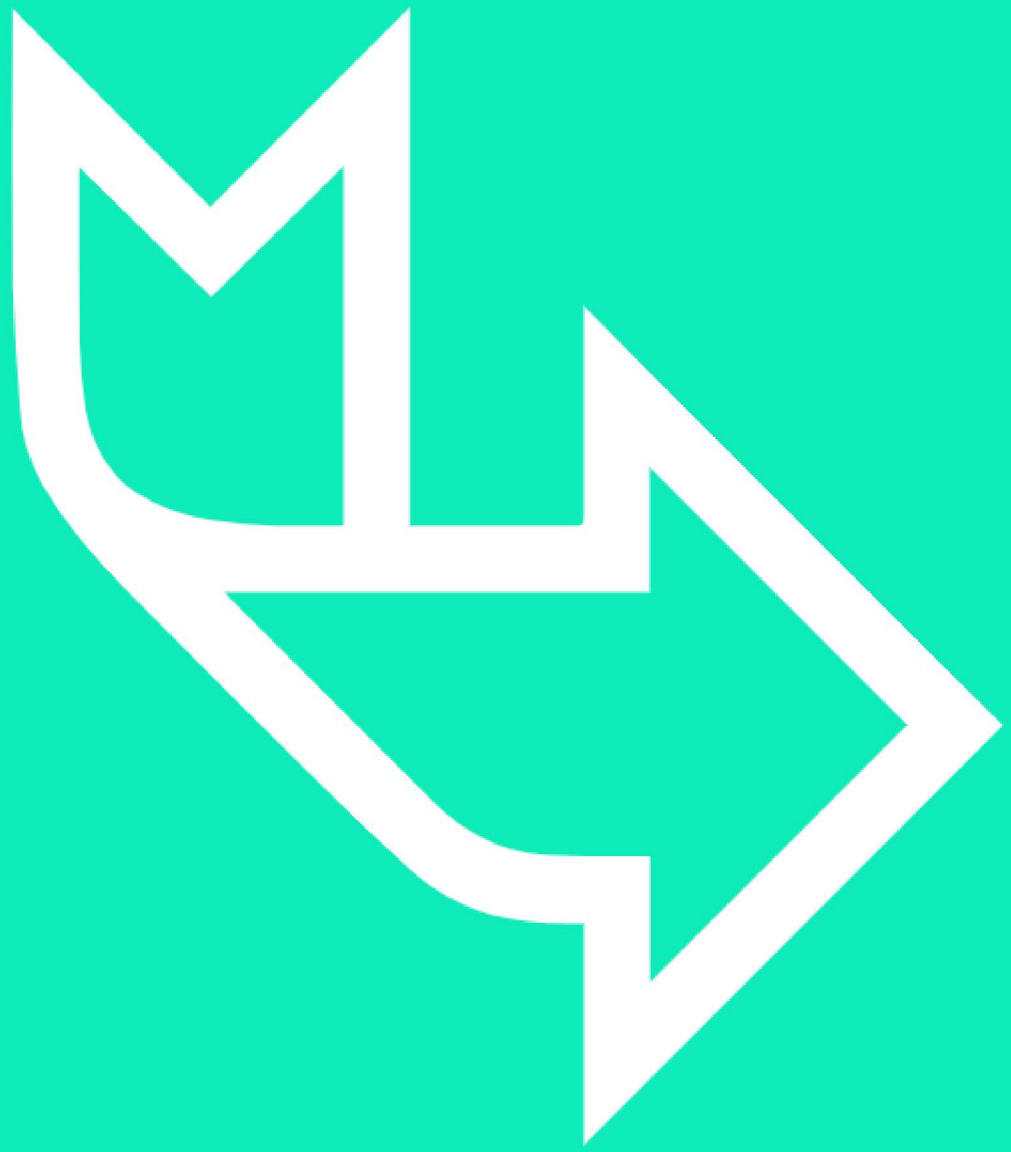
```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>React App</title>
  </head>

  <body>
    <div id="root">
      <!-- App renders in here! -->
    </div>
  </body>
</html>
```

Just like with vanilla JavaScript, you have a webpage which renders the JS in a browser. React is no different .



index.js

```js
import React, { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'

import App from './App'

// Finds the HTML element with an id of "root"
const rootElement = document.getElementById('root')

// Renders the App component inside of the root element.
const root = createRoot(rootElement)

root.render(
  <StrictMode>
    <App />
  </StrictMode>
)
```



Quick Lab 1

Install and edit your first React App

Components in React

Overview

Components are the building blocks of any React application. They are snippets of code that are called into render when required by the SPAs we create.

Generally, it's a good idea to split the components up and then call them into App.js when required.

The easiest way to learn this is to do it.

Start by creating a new file called MainContent.jsx in your src folder:

Overview

Inside this we need to create a function that returns some JSX.

This is an arrow function that renders when called. The **return** is what is passed to the index.html page when all components are rendered to **App.jsx.**

```
const MainContent = () => {
    return (
        <div className="content">
            <h1>Welcome to my Webpage</h1>
            <p>It's very easy to pick up</p>
        </div>
    );
}

export default MainContent;
```

Don't forget to export at the bottom – just like a JS module.

Overview

Inside **App.js** import this module at the top:

```
1  import MainContent from './MainContent';
```

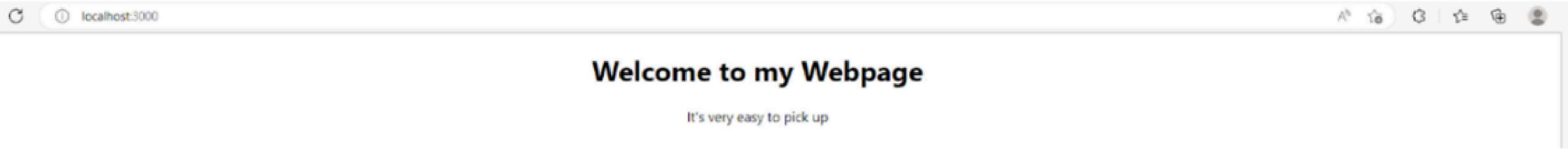
Then call the function to be rendered, replacing the **<h1> tag**

```
1  import MainContent from './MainContent';
2  import './App.css';
3
4  function App() {
5    return (
6      <div className="App">
7        <MainContent />
8      </div>
9    );
10 }
11
12 export default App;
13
```

Don't forget to close the function call!

Overview

Once you save it, your webpage should re-render to this:



All content in React should be in its own component **.jsx** file

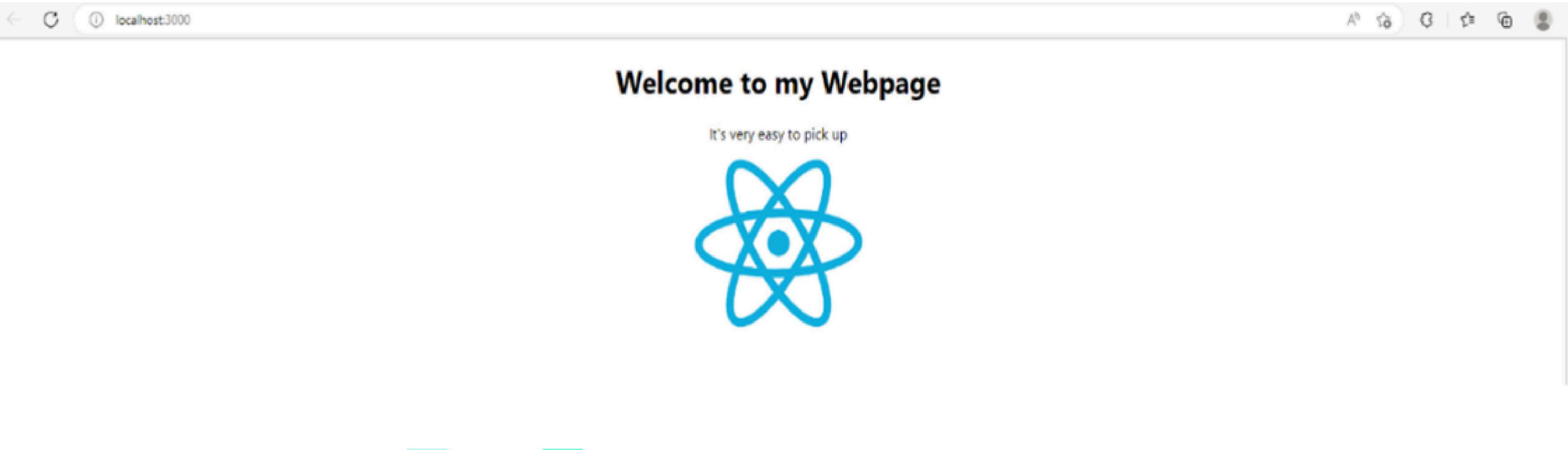
Overview

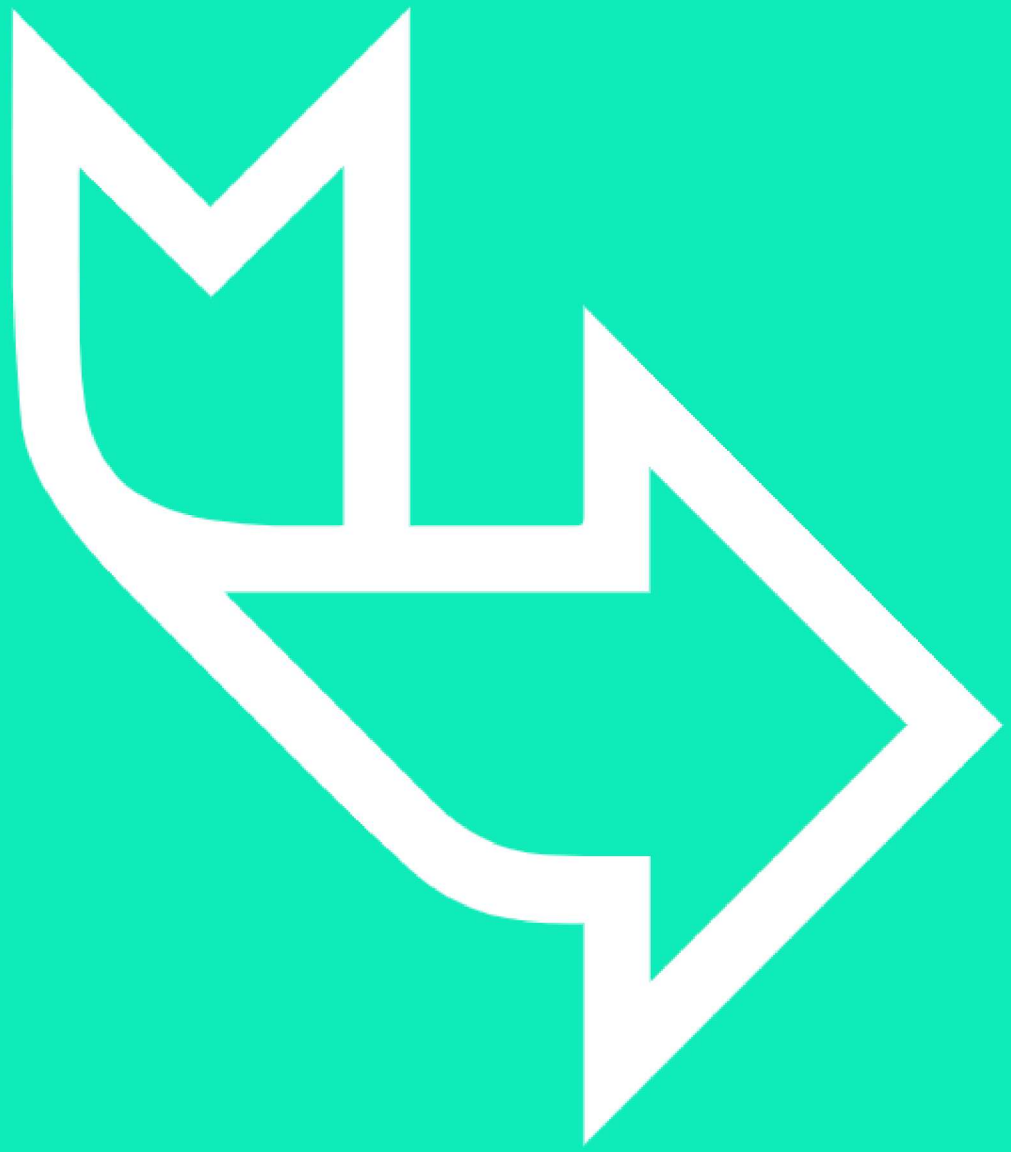
More Components and more imports build our page:


```jsx
const MoreContent = () => {
    return (
        <div className="image">
            <img src="https://i.pinimg.com/originals/84/b1/06/84b1065e798f61aa80b8670a4b6fbb4d.png"
                width="200px"
            />
        </div>
    );
}

export default MoreContent;
```

```jsx
import MainContent from './MainContent';
import MoreContent from './MoreContent';
import './App.css';

function App() {
  return (
    <div className="App">
      <MainContent />
      <MoreContent />
    </div>
  );
}

export default App;
```

Overview


localhost:3000
Welcome to my Webpage
It's very easy to pick up

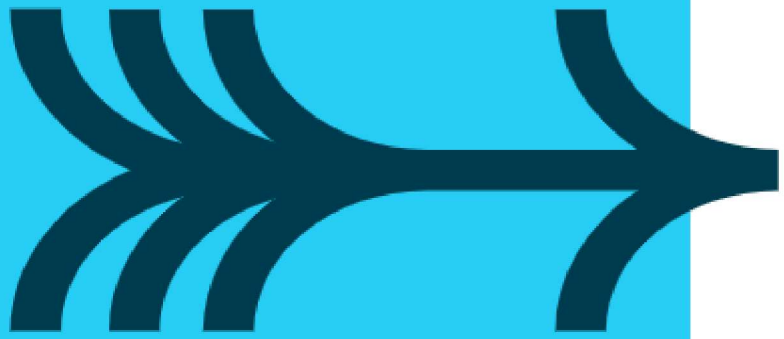


Quick Lab 2

Create your first components

Embedding dynamic content in JSX

EXPRESSIONS



One of the major benefits of using React and JavaScript is the ability to have variables and content which are dynamic.

You will remember that in JS, we can assign a const or let and then use that value in expressions { } within the code block.

Variables and constants are usually created above the return and then used in the return itself.

EXPRESSIONS

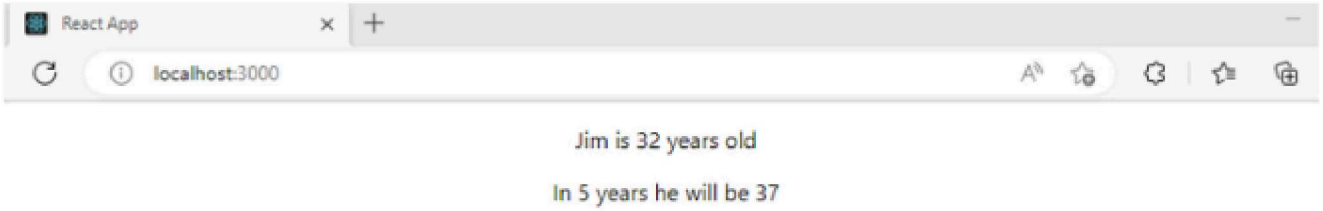
This will render the value of the constant above in the return. It also works with numbers.

```
1   const Exp = () => {
2       const username = "Jim";
3       const age = 32;
4
5       return (
6           <div className="output">
7               <p>{ username } is { age } years old</p>
8           </div>
9       );
10  }
11
12  export default Exp;
```

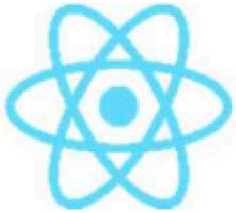
EXPRESSIONS

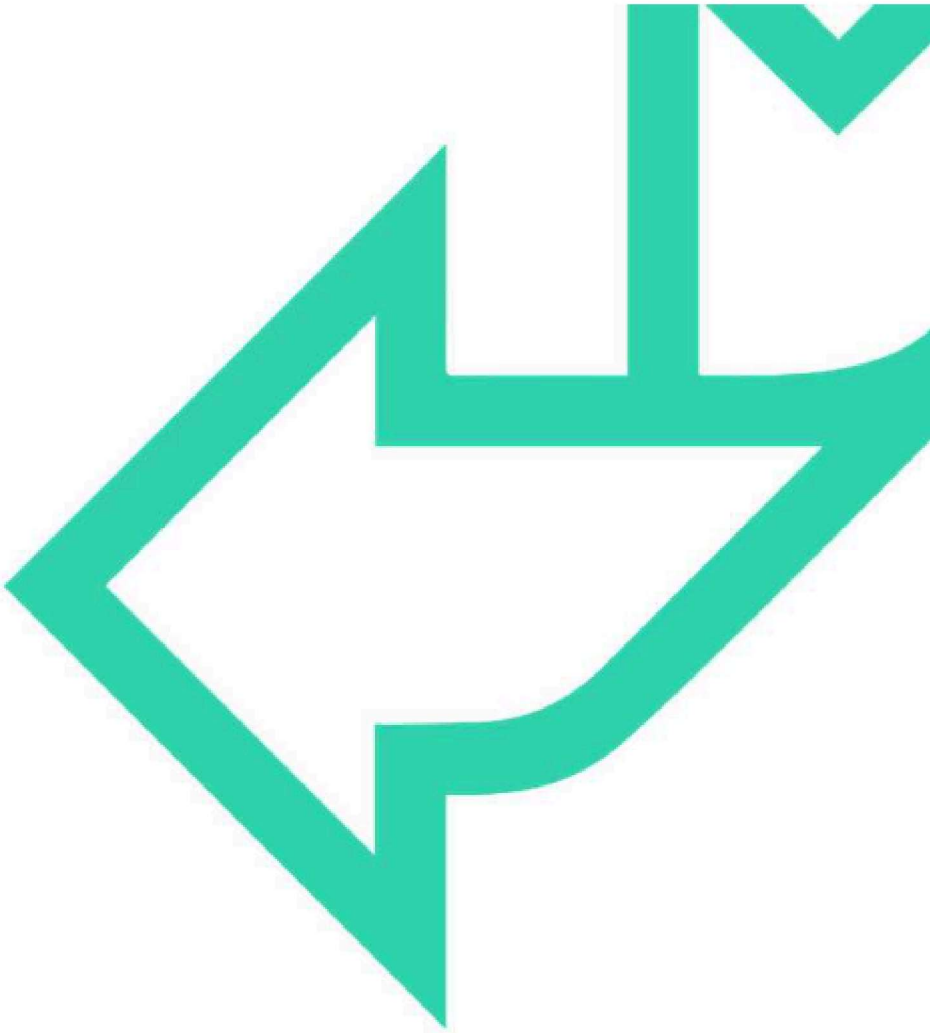
Like other Expressions, you can perform calculations within the curly braces.

```
const Exp = () => {
    const username = "Jim";
    const age = 32;

    return (
        <div className="output">
            <p>{ username } is { age } years old</p>
            <p>In 5 years he will be { age + 5 }</p>
        </div>
    );
}

export default Exp;
```



QA POWERING POTENTIAL





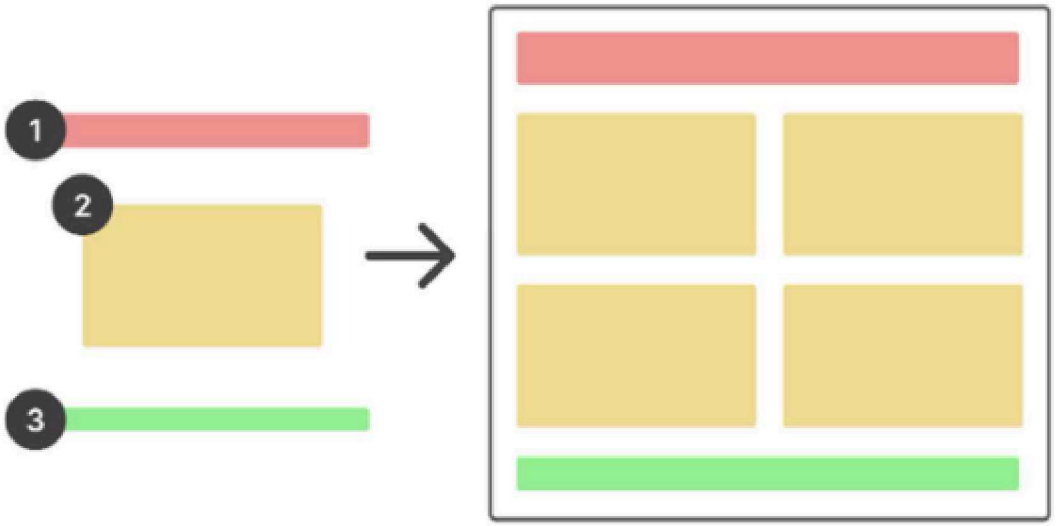
HANDS-ON WITH REACT

QA

REACT OVERVIEW

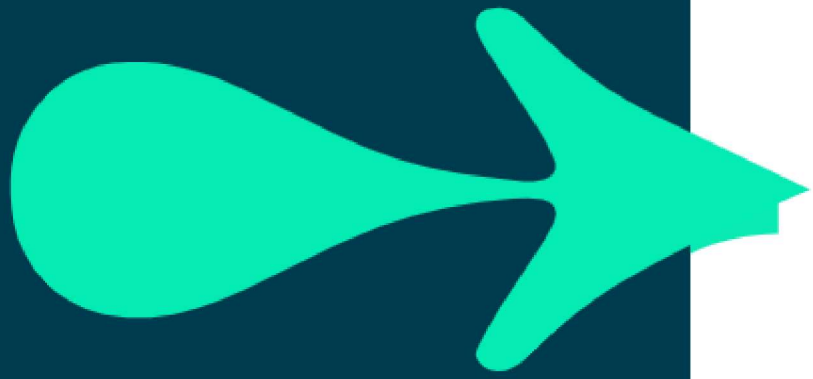
What is React?

- **A JavaScript library** that allows you to build user interfaces (UIs) out of a hierarchy of 'components'.
- **Widely used –** Facebook, Instagram, Reddit, Twitter, Netflix, Airbnb, BBC, and many more use React for their user interfaces.



QA

REACT OVERVIEW



Why React?

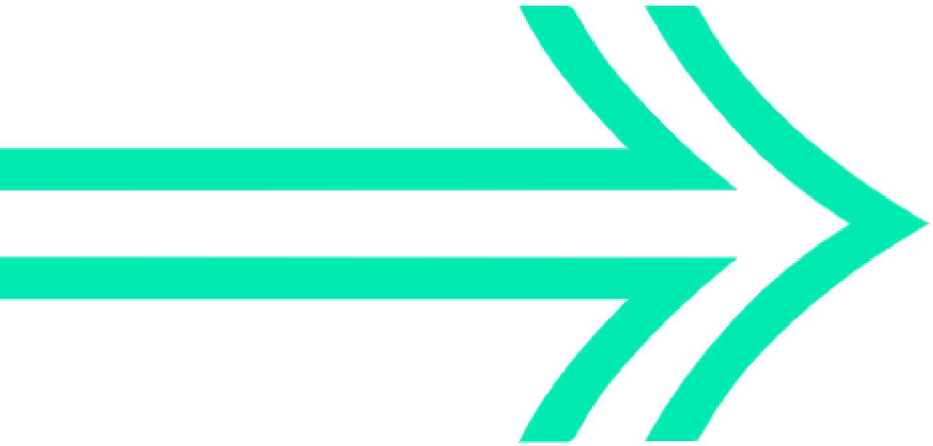
- It's **approachable**. Anyone with an understanding of JavaScript and HTML can pick up React.
- It's **composable**. Building your UI with reusable components makes your code simpler, more logical, easier to read, and less repetitive.
- Creating **dynamic, interactive applications** becomes much easier thanks to React features like state.
- Its **rich ecosystem** of community-created tools helps to make your life easier.

PROPS

React props

Props are React's way of passing information to a component.

Like how an <img> tag in HTML can accept attributes like "src" and "width", React components can accept data in the form of props.

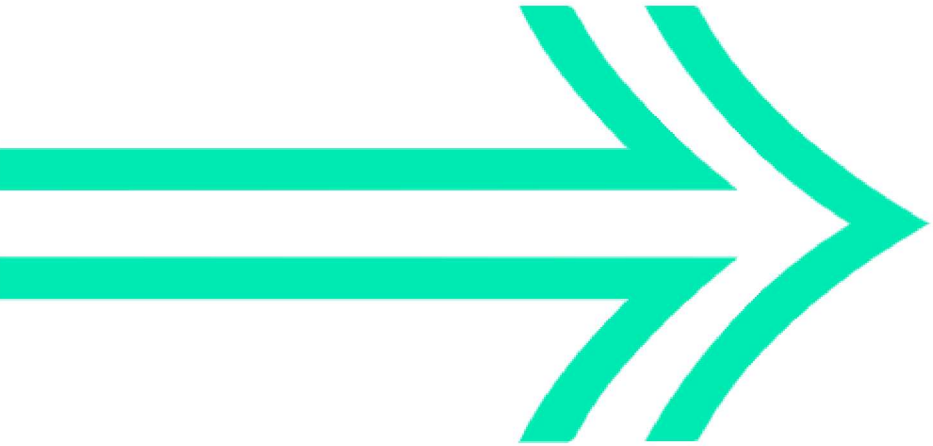


Props are incredibly useful for creating **reusable components**.

React props

I can create a component which can be reused time and time again.

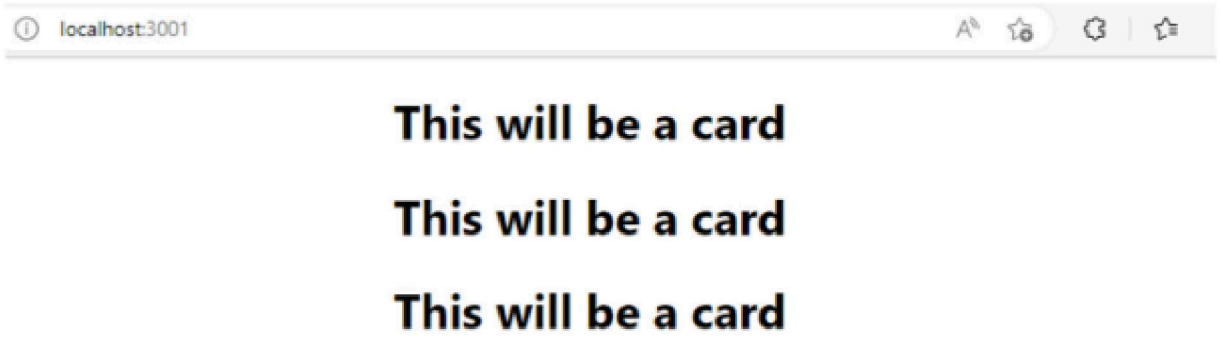
```
1   const Card = () => {
2       return (
3           <h1>This will be a card</h1>
4       );
5   }
6
7   export default Card;
```



```
import Card from './components/Card';
import './App.css';

function App() {
  return (
    <div className="App">
      <p><Card /></p>
      <p><Card /></p>
      <p><Card /></p>
    </div>
  );
}

export default App;
```

React props

This creates a predictable output.

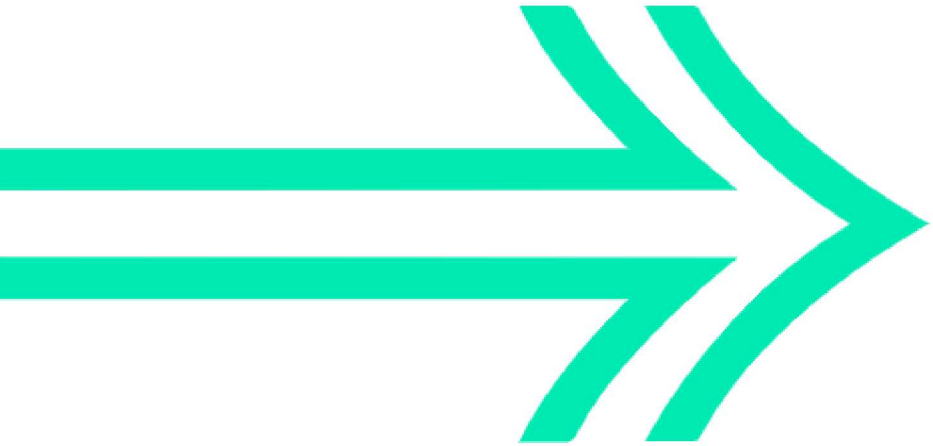


React props

Props allow us to pass data to a child component and then have it used and re rendered based on the content we have given it.

For example, I can add a parameter to the function and then use it in the return, just like JS.

```
1   const Card = ( {name} ) => {
2       return (
3           <h1>This will be a card from {name}</h1>
4       );
5   }
6
7   export default Card;
```

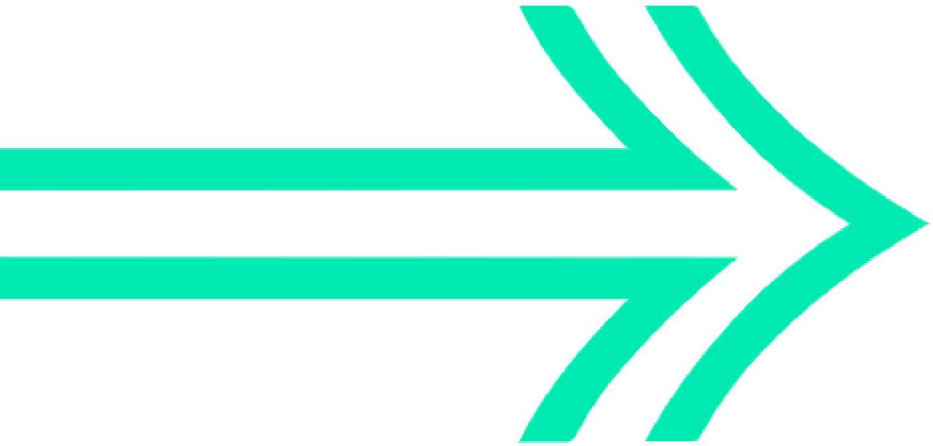


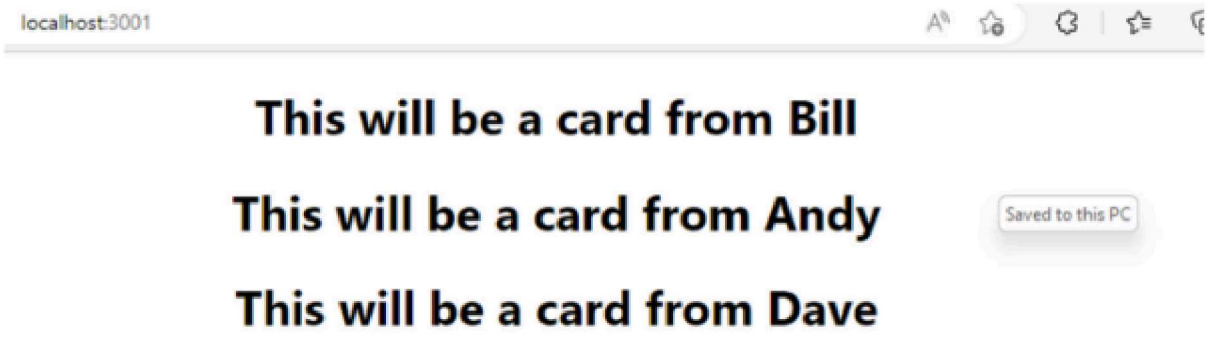
This now requires information to be passed with the call. It will be deconstructed like a map.

React props

Adding this as Props to the call allow it to be returned and rendered in the App:

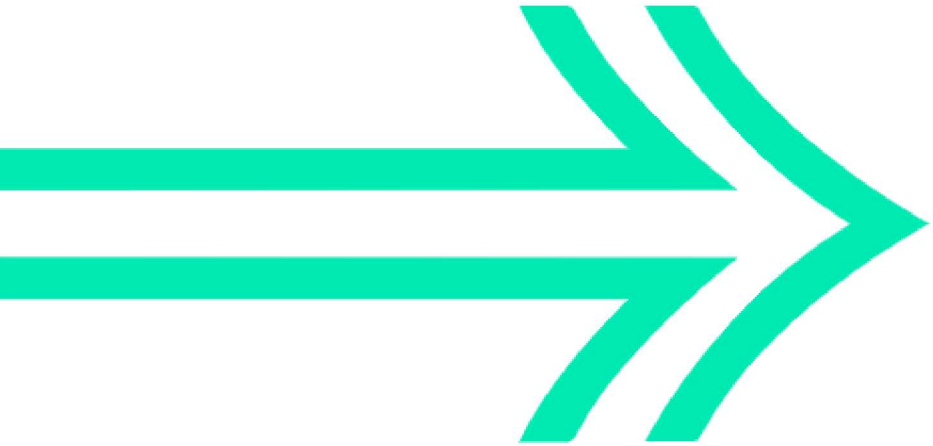
```
import Card from './components/Card';
import './App.css';

function App() {
  return (
    <div className="App">
      <p><Card name="Bill" /></p>
      <p><Card name="Andy" /></p>
      <p><Card name="Dave" /></p>
    </div>
  );
}

export default App;
```





React props

We can now change this content and pass more than one **Prop** to the component call.



```jsx
const Card = ( {name, job, department} ) => {
    return (
        <div className="card">
        <h1>{name}</h1>
        <h2>{job}</h2>
        <h2>{department}</h2>
        </div>
    );
}

export default Card;
```

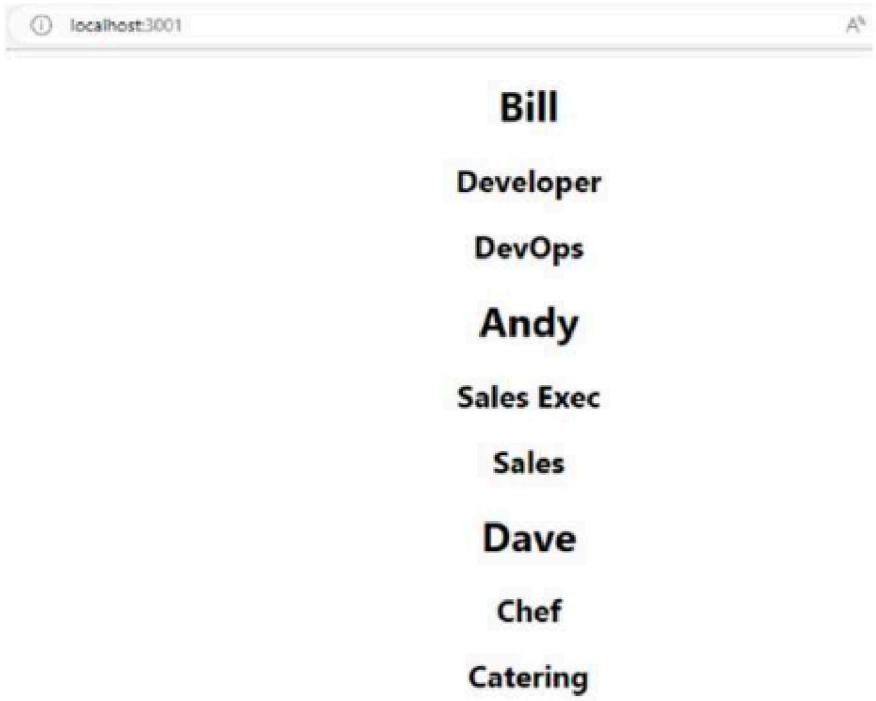
React props

We can now change this content and pass more than one **Prop** to the component call.

```
import Card from './components/Card';
import './App.css';

function App() {
  return (
    <div className="App">
      <p><Card name="Bill" job="Developer" department="DevOps" /></p>
      <p><Card name="Andy" job="Sales Exec" department="Sales"/></p>
      <p><Card name="Dave" job="Chef" department="Catering"/></p>
    </div>
  );
}

export default App;
```

React props

We can now change this content and pass more than one **Prop** to the component call.



React props

Let's add an image and edit some CSS:

```
const Card = ( {name, job, department} ) => {
    return (
        <div className="card">

            <img src="https://pedodoctor.com/custom/images/dr-jeffries.jpg"
             width="100px"
             />
            <h1>{name}</h1>
            <h2>{job}</h2>
            <h2>{department}</h2>

        </div>
    );
}

export default Card;
```

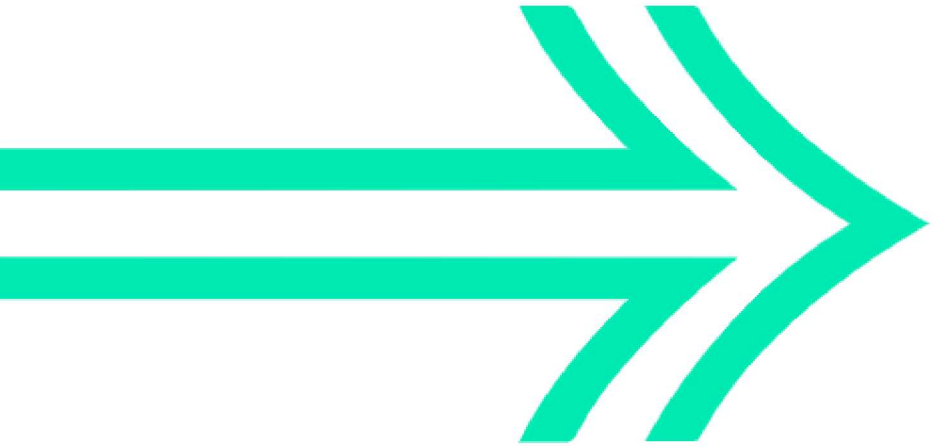
React props

Let's add an image and edit some CSS:

App.css -

```css
.App {
  text-align: center;
}

.card {
  padding: 5px;
  border-bottom: 1px lightgray solid;
  max-width: 600px;
  margin: auto;
  text-align: left;

}
 img {
  float: right;
  padding: 40px;
 }
```

React props



Gives us this:

Bill

Developer

DevOps

Andy

Sales Exec

Sales

Dave

Chef

Catering

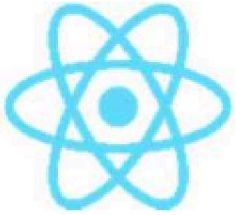
QA React props

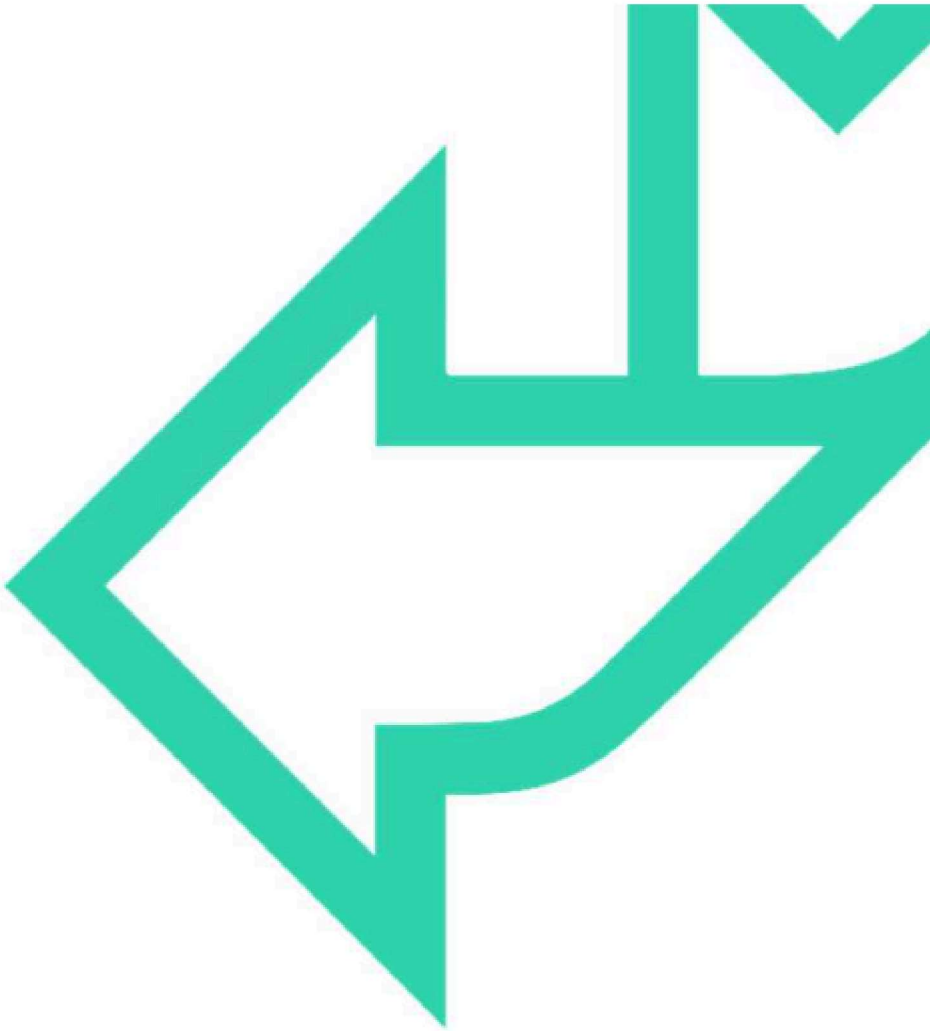
We can also pass the **Props** in and deconstruct them within the function itself.

```
const Card = (props) => {
    const name = props.name;
    const job = props.job;
    const dep = props.department

    return (
        <div className="card">

            <img src="https://pedodoctor.com/custom/images/dr-jeffries.jpg"
            width="100px"
            />
            <h1>{name}</h1>
            <h2>{job}</h2>
            <h2>{dep}</h2>

        </div>
    );
}

export default Card;
```


QuickLab 3
Working with Props in React

QA POWERING POTENTIAL

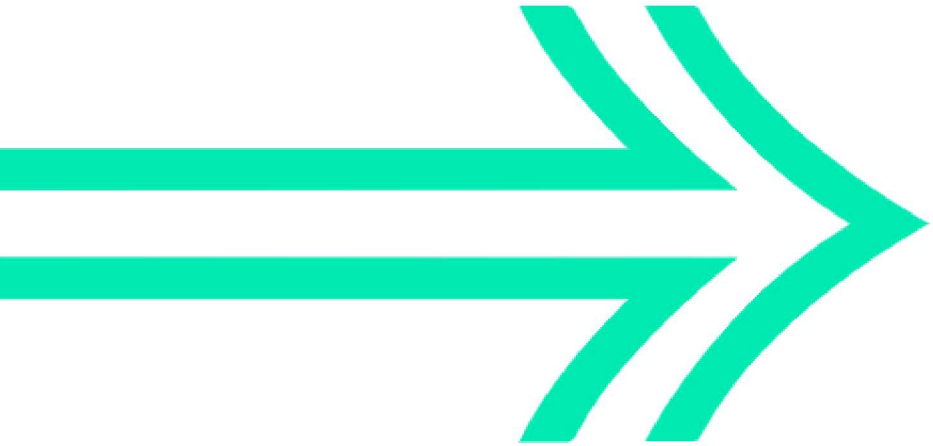




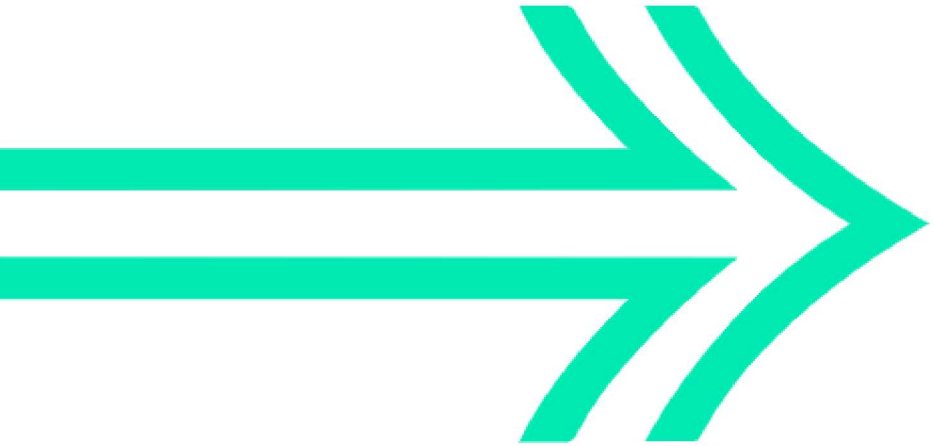
HANDS-ON WITH REACT

Conditional rendering

We often need **change how a component is displayed** depending on certain conditions.



How to conditionally render content



To conditionally render information is one of the staples of being able to keep a website up to date.

We have no idea when data may change, but when it does, we need our creations to immediately reflect the change.

How to conditionally render content

We need to add a new parameter to the list of props accepted by the Card component.

isActive will be a Boolean value that we can then add or remove content to the webpage.

We also need to include it in the Card call in **App.js.**

```
import Card from './components/Card';
import './App.css';

function App() {
  return (
    <div className="App">
      <p><Card name="Bill" age="35" role="DevOps" isActive={true} /></p>
      <p><Card name="Andy" age="27" role="Sales" isActive={false}/></p>
      <p><Card name="Dave" age="64" role="Catering" isActive={true}/></p>
    </div>
  );
}
export default App;
```

How to conditionally render content

We need to add a new parameter to the list of props accepted by the Card component.

isActive will be a Boolean value that we can then add or remove content to the webpage.

We also need to include it in the Card call in **App.js.**

```
const Card = ({name,age,role, isActive}) => {


    return (
        <div className="card">

            <img src="https://pedodoctor.com/custom/images
            width="100px"
            alt=""
            />
```

How to conditionally render content

This can be done multiple ways:

1) Render entire card based on Active Status

```
const Card = ({name,age,role, isActive}) => {
    if(isActive){
        return (
                <div className="card">
                    <img src="https://pedodoctor.com/custom/images/dr-jeffries.jpg"
                    width="100px"
                    alt=""
                    />
                    <h1>{name} ✔</h1>
                    <h2>{age}</h2>
                    <h2>{role}</h2>
                </div>
            );
    } else {
        return (
            <div className="card">
                <img src="https://pedodoctor.com/custom/images/dr-jeffries.jpg"
                width="100px"
                alt=""
                />
                <h1>{name} ❌</h1>
                <h2>{age}</h2>
                <h2>{role}</h2>
            </div>
        );
    }
}

export default Card;
```

How to conditionally render content

This can be done multiple ways

1) Ternary operator

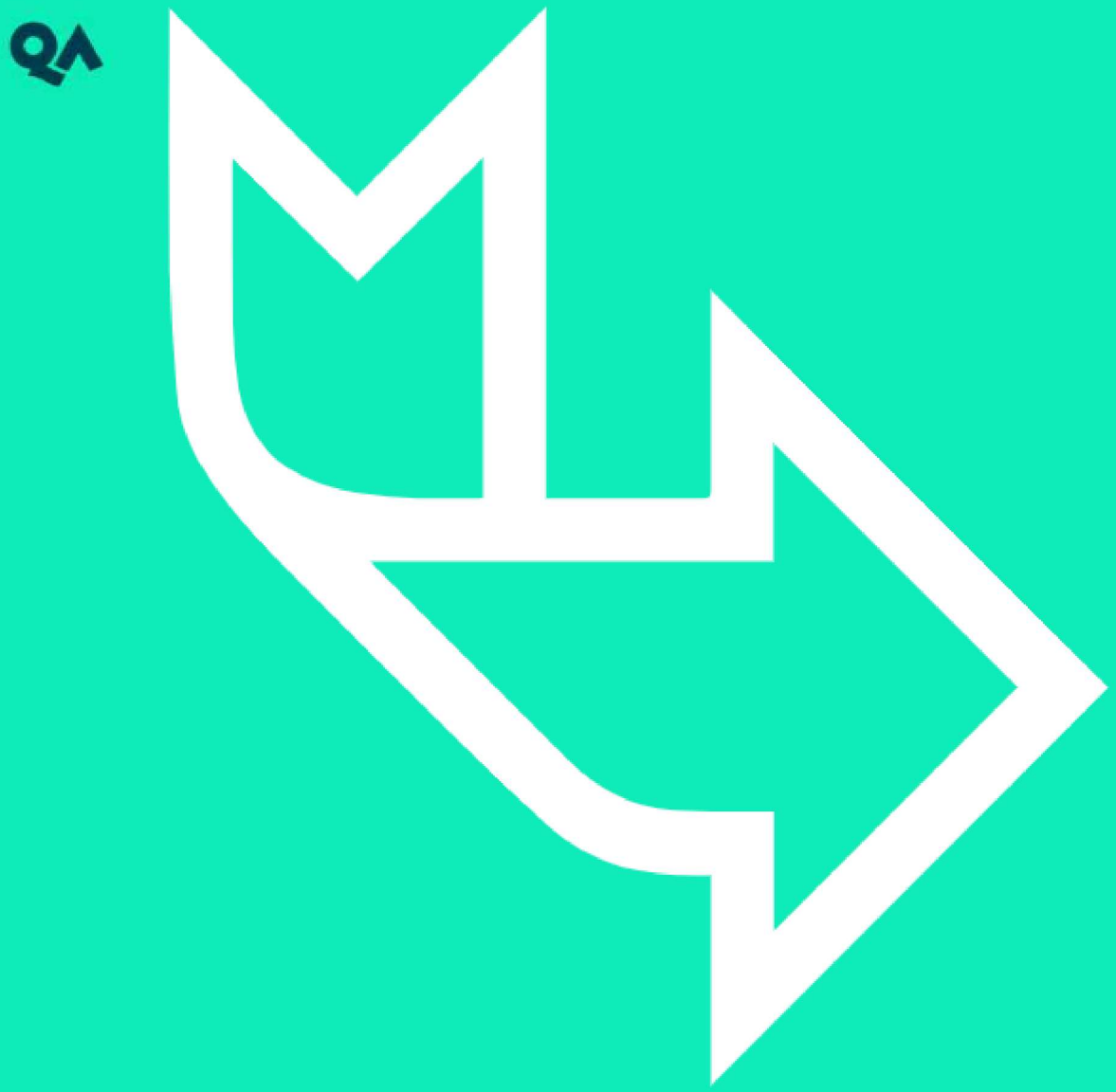
```
return (
        <div className="card">
            <img src="https://pedodoct
            width="100px"
            alt=""
            />
            <h1>{name}</h1>
            <h2>{age}</h2>
            <h2>{role}</h2>

            {
            isActive
            ? <h2>Actve</h2>
            : <h2>Inactive</h2>
            }

        </div>
```

How to conditionally render content

This can be done multiple ways:

1) Ternary operator, Logical && Operator

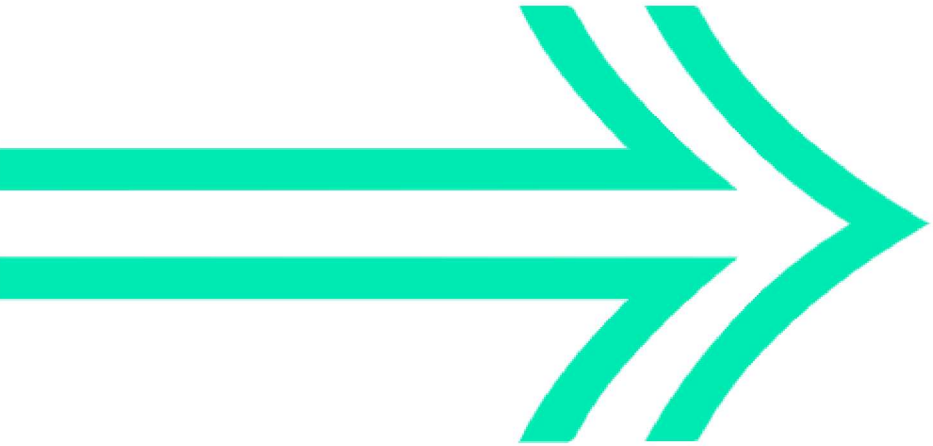
```
const Card = ({name,age,role, isActive}) => {

        return (
                <div className="card">
                    <img src="https://pedodoctor.com/custom/images/dr-jeffries.jpg"
                    width="100px"
                    alt=""
                    />
                    <h1>{name} {isActive && '✔'} </h1>
                    <h2>{age}</h2>
                    <h2>{role}</h2>


                </div>
            );

}

export default Card;
```



QuickLab 4

Conditional Rendering

Rendering lists



Displaying many similar components from a collection of data is something that happens all the time.

Let's look at how to transform an array of data into an array of components.

Rendering lists

There is an array of data dropped into the **App.js** file. This is JSON data that you learned about earlier in this section.

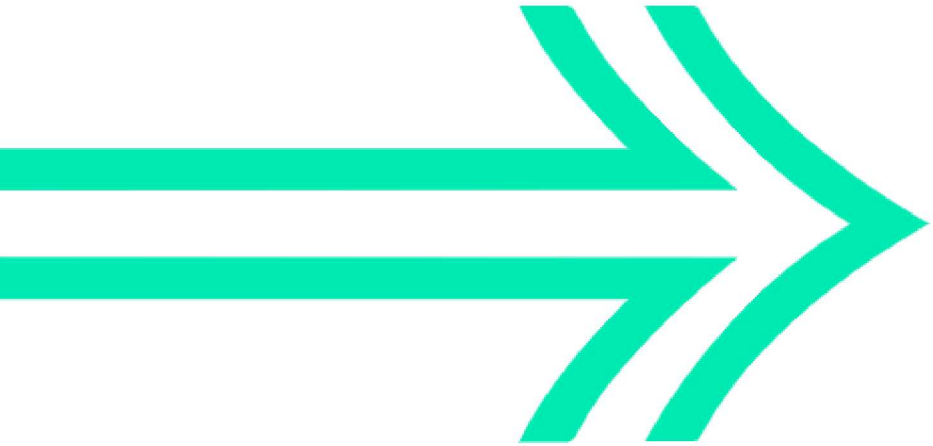
```
function App() {
  const blogs = [
    {title: 'My first Blog', head: 'This is first', author: 'Dave', id: 1},
    {title: 'Welcome', head: 'Come along', author: 'Will', id: 2},
    {title: 'My first Blog', head: 'This is first', author: 'Frank', id: 2}
]
```

I can use JS map() tools to take this information from the **const** called blogs and output it onto a page.

Rendering lists

The return takes the blogs and passes them as a callback function. This allows us to simply deconstruct the array.

```
return (
  <div className="App">
    {blogs.map((blog) =>(
      <div className="blog-preview" key={blog.id}>
        <h2>{ blog.title }</h2>
        <p>Written by {blog.author}</p>
      </div>
    ))}
  </div>
);
```



Notice the dot(.) notation. This is taking each blog from blogs and doing the same thing to it.

Just like for OR foreach

Rendering lists

However, it is likely that you will be collecting data from external sources such as APIs, Databases etc and it is important to understand how to go about doing this in React.

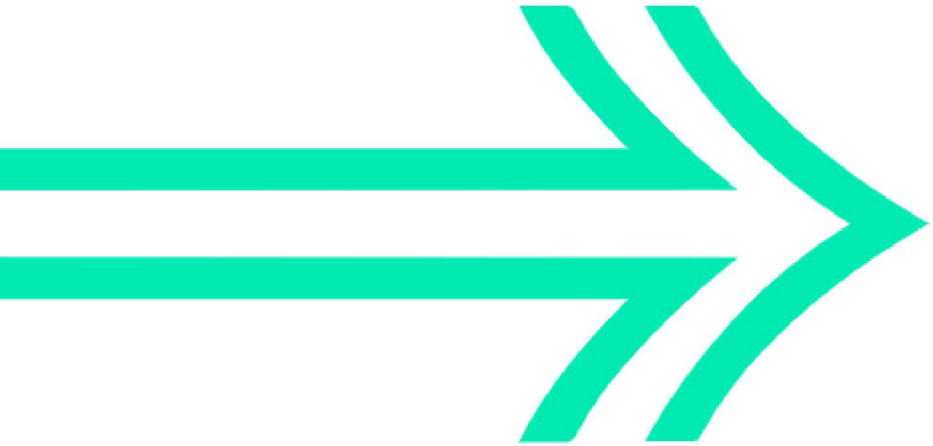
A file has been created with some data relating to some fruit.

```
[
    {
      "id": 1,
      "symbol": "🍎",
      "name": "Apple",
      "price": 0.3
    },
    {
      "id": 2,
      "symbol": "🍍",
      "name": "Pineapple",
      "price": 1
    },
```

Rendering lists

Like before, I will create a card that will help with styling the components and put it in a separate JSX file.

```jsx
import React from "react";

function ItemCard({ symbol, name, price }) {
    return (
        <div className="item-card">
            <div className="symbol">{symbol}</div>
            <h3>{name}</h3>
            <p>£{price.toFixed(2)}</p>
        </div>
    )
}
export default ItemCard;
```

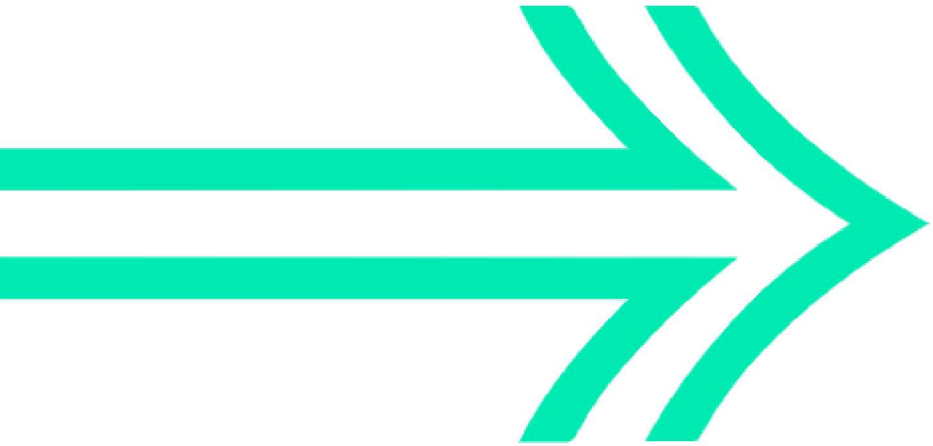


This card has been set up to accept the content of the JSON file and will be passed the data as props from **App.js.**

Rendering lists

Firstly, we need to import the data to react from the JSON file.

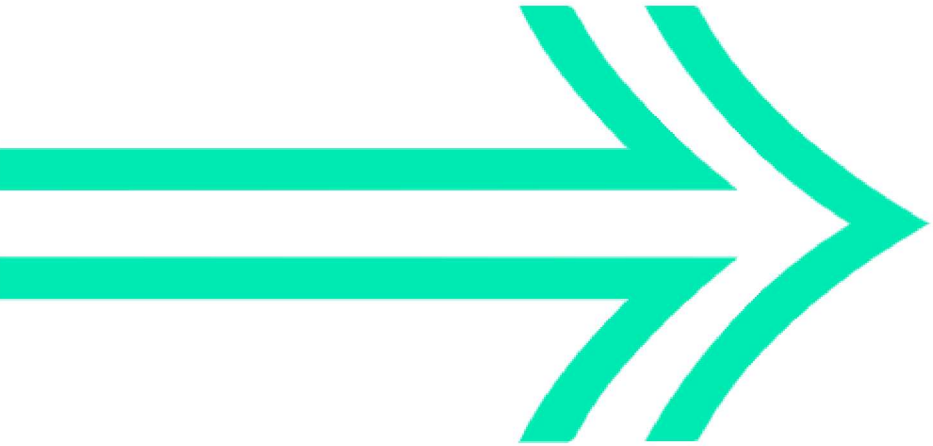
```
import ItemCard from './components/items';
import itemsData from './itemsData.json';
```



This takes the entire JSON file and drops it into a variable called itemsData. It can then be mapped through.

```
function App() {

  return (
    <div className="App">
      <main>
        <h1>React Fruit Market</h1>
        <div className="item-grid">
          {itemsData.map((item) => (
            <ItemCard
            key={item.id}
            symbol={item.symbol}
            name={item.name}
            price={item.price}
            />
          ))}
        </div>
      </main>
    </div>
```

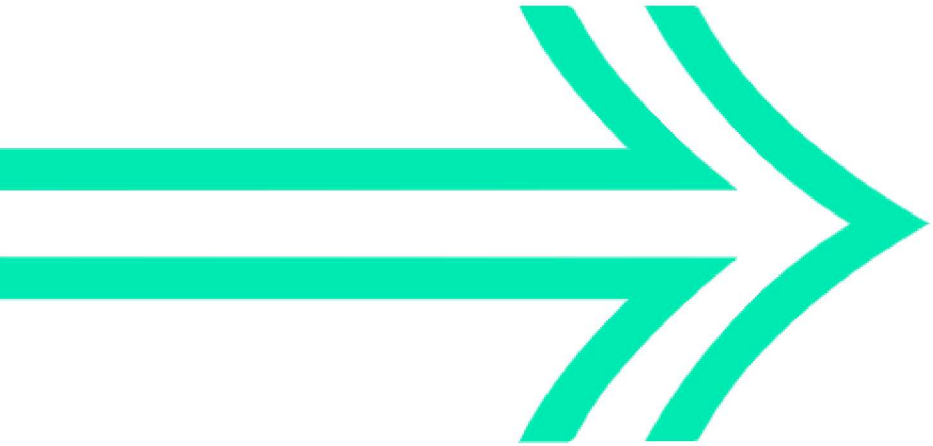
Rendering lists



So, the data is in there nicely, but we need to apply some stylings to ensure that our website looks the part.

Rendering lists

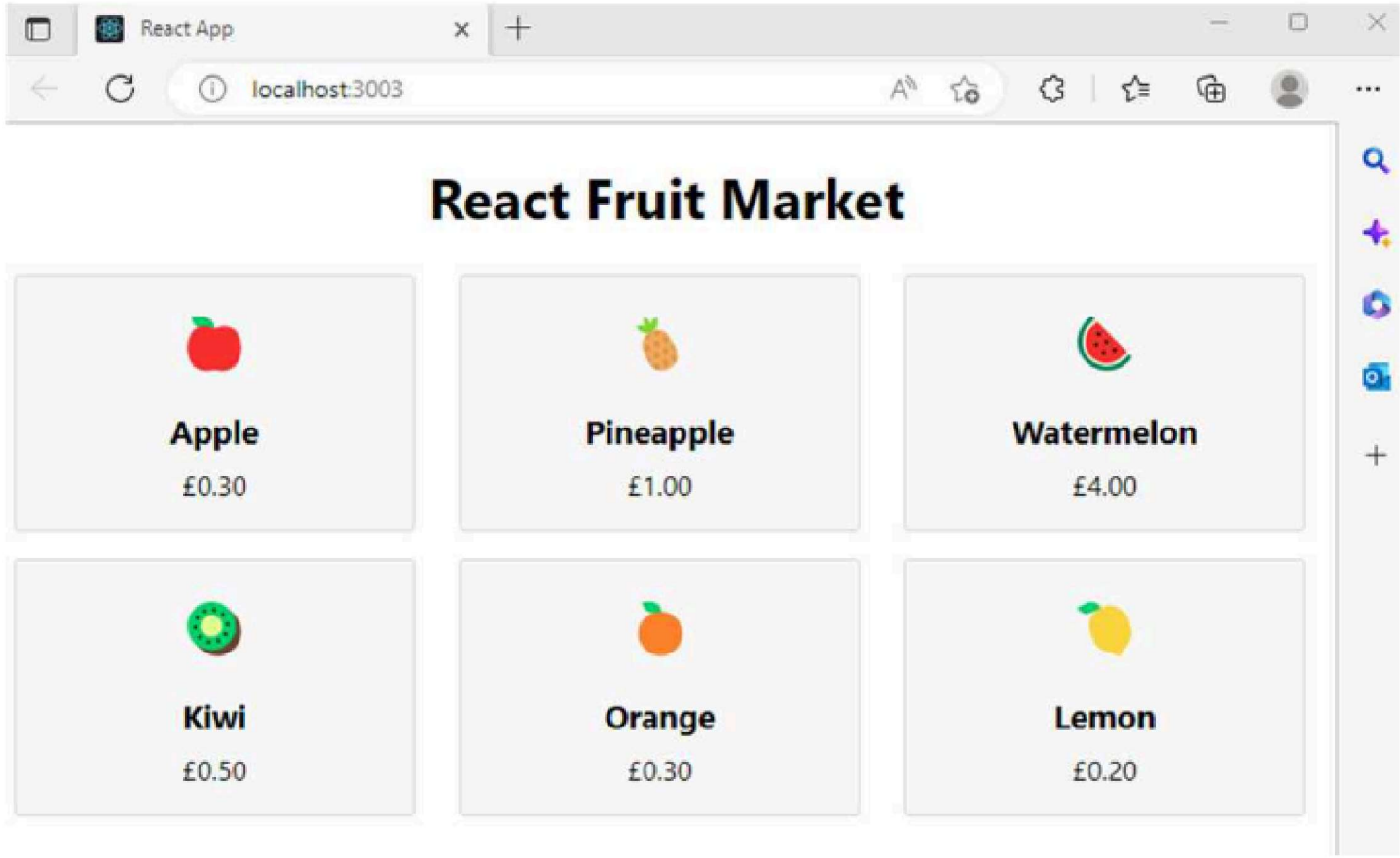
App.css Stylings will turn these into cards, similar to bootstrap and other library styles.



```css
h1 {
  margin-bottom: 1.5rem;
}

.items-grid {
  display: flex;
  flex-wrap: wrap;
  gap: 1rem;
}

.item-card {
  margin-left: 10px;
  width: 200px;
  padding: 1rem;
  background-color: rgb(248, 248, 248);
  border-radius: 3px;
  border: solid 1px rgb(220, 220, 220);
}

.item-card .symbol {
  font-size: 2rem;
  text-align: center;
}

.item-card h3 {
  margin-bottom: 0.5rem;
}

.item-card p {
  margin-left: 0;
}
```

Rendering lists

This results in data being pulled from a file and rendered on our webpage.




QuickLab 5
Rendering External Data

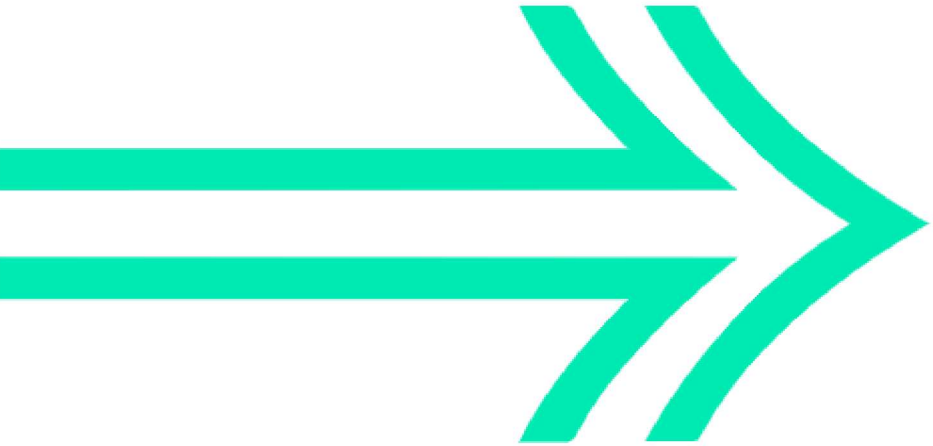
React Router

Most websites have different **routes**.

React Router provides lots of useful routing-related features, including a simple way to display **different content depending on which route** is currently active.

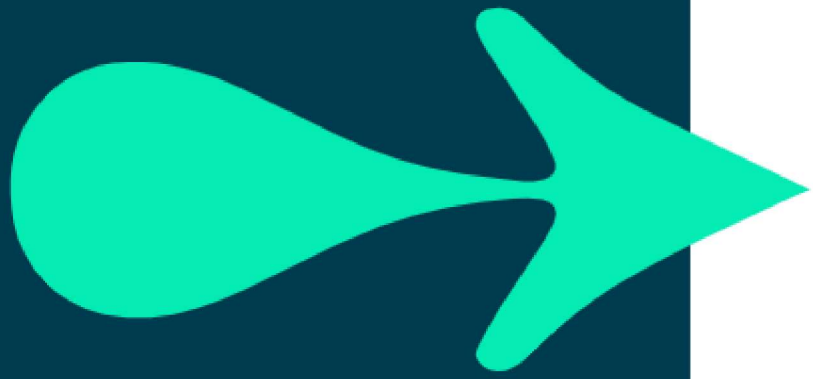
www.example.com/ → <HomePage />

www.example.com/about → <AboutPage />



This facilitates the creation of **single-page applications**.

SINGLE-PAGE APPS

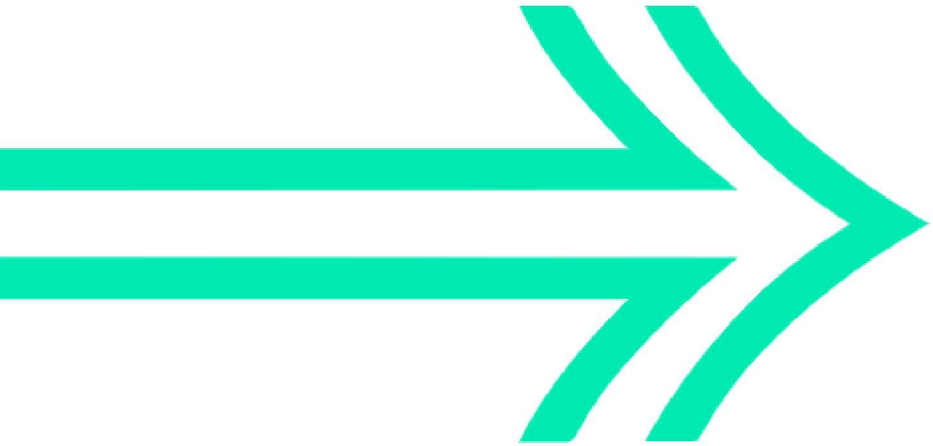


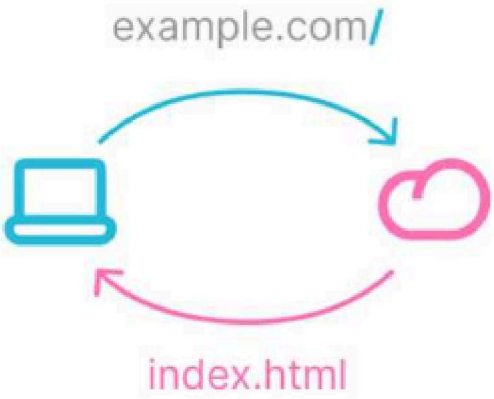
So, what is a single-page application?

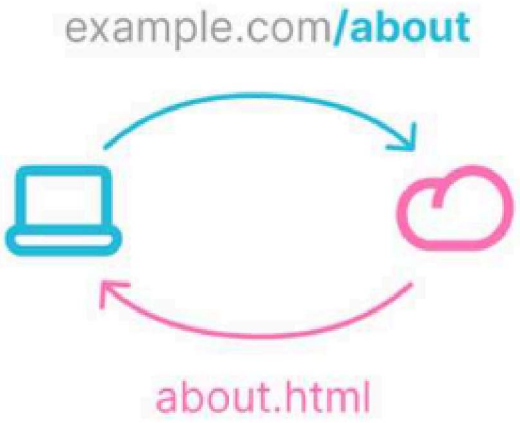
Contrary to what the name suggests, a single-page app (or SPA) is not just a website where all the content is contained on one long page.

Let's look at the difference between a traditional **multi-page** site and a **single-page** application.

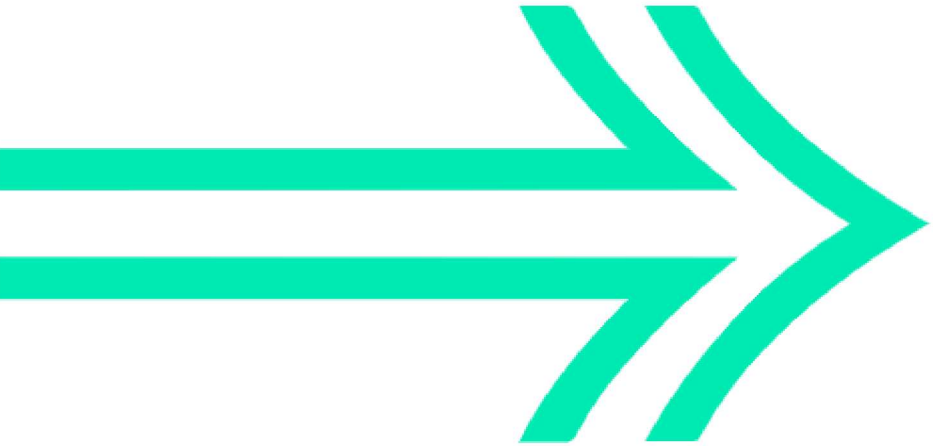
Traditional (multi-page) website

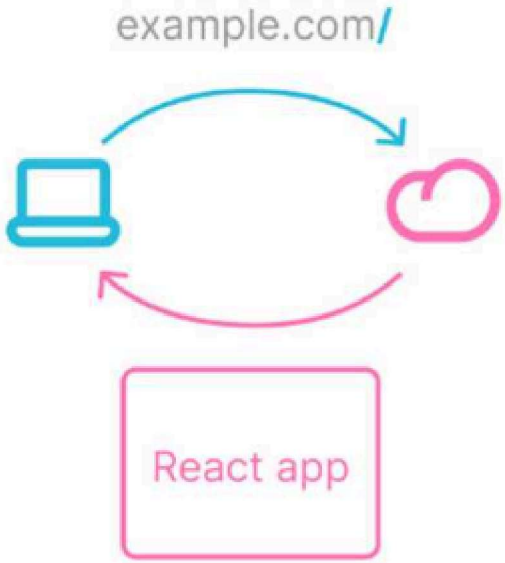


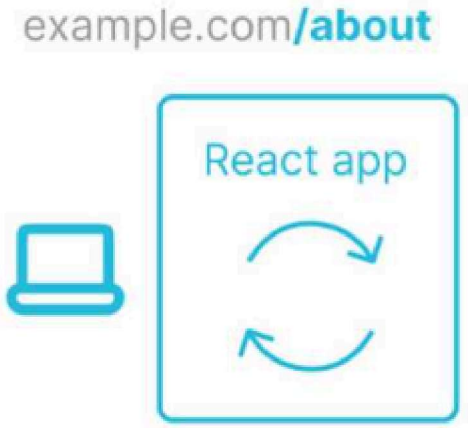




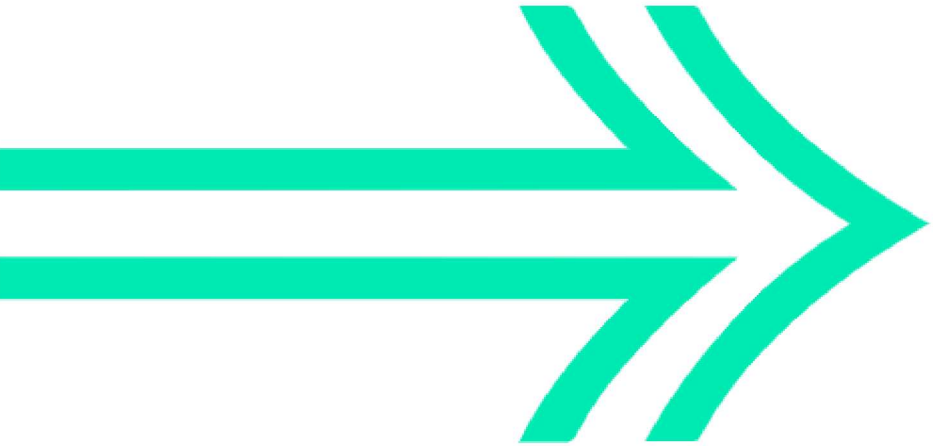
Single-page application







React Router



Now that we've covered the concept of client-side routing, let's see it in practice.

For Routing to work in react, You need to run some npm installs first:

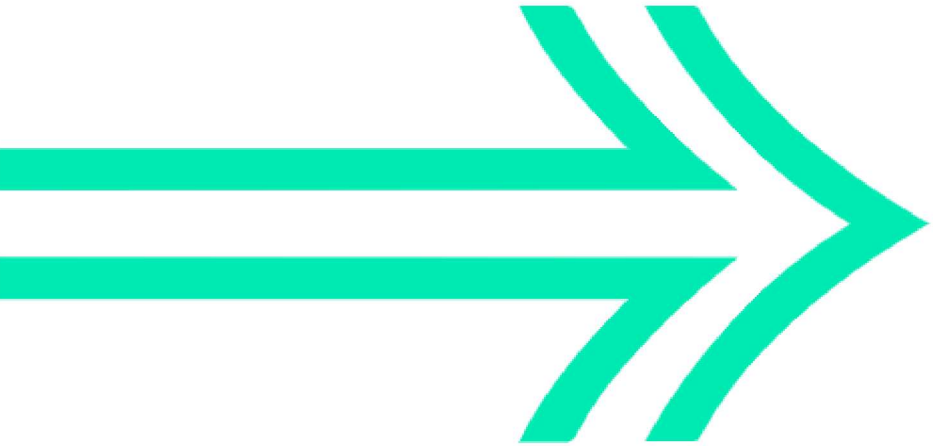
```
npm install --save react-router
```

And:

```
npm i -S react-router-dom
```

These allow up to set up pages for routing in SPA set up just discussed

React Router

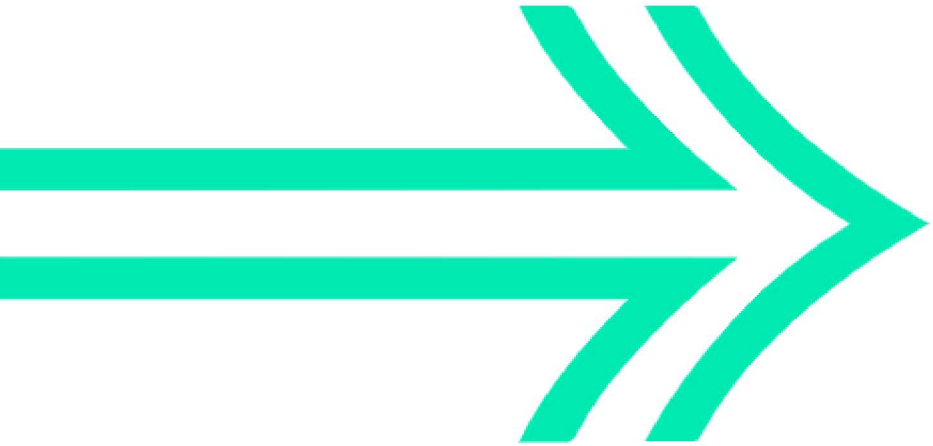


It is good practice to set up a nav bar component which will appear on each of our pages. This can be done in the usual way.

On thing that will change is that we will need to import { Link } from the react-router-dom to tell react that we want to render different pages to the app.

When we then use the <Link> tag, we specify where we want the link to take us to. For example, “/” is the home page and “/about” would be https://127.0.0.1/about

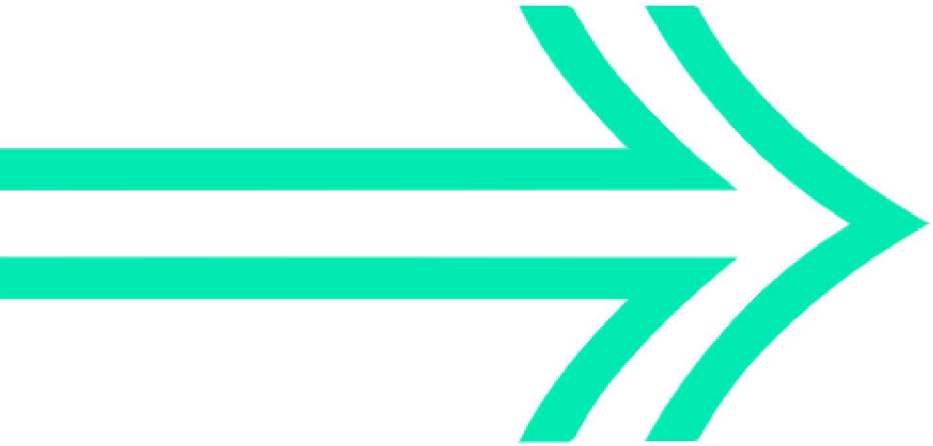
React Router



This gives us two links with predefined destinations:

```
import React from "react";

import { Link } from 'react-router-dom'

export default function Navbar() {
    return (
        <nav>
            <h1>React App</h1>
            <ul>
                <li>
                    <Link to="/">Home</Link>
                </li>
                <li>
                    <Link to="/about">About</Link>
                </li>
            </ul>
        </nav>
    )
}
```

React Router



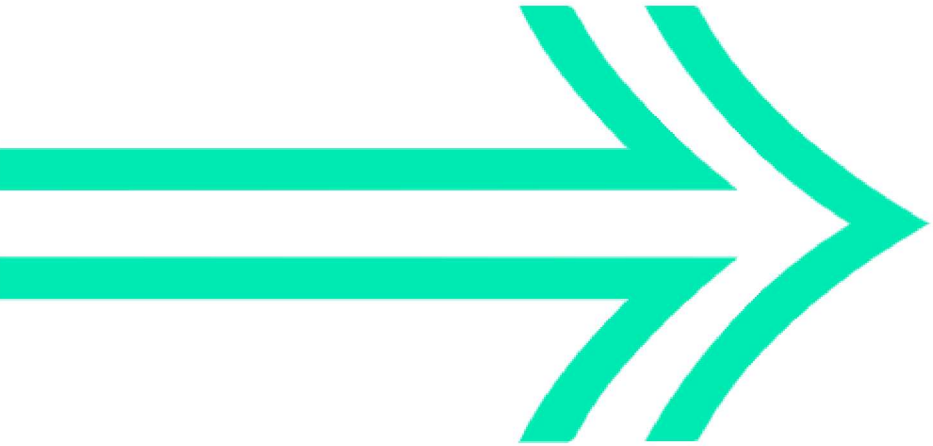
We can render two simple components for testing:

```jsx
export default function AboutPage() {
    return (
        <div>
            <h2>This is the about Page</h2>
            <p>Some helpful information</p>
        </div>
    )
}
```

```jsx
export default function HomePage() {
  return (
    <div>
      <h2>This is the home page.</h2>
      <p>Welcome to this website!</p>
    </div>
  )
}
```

React Router

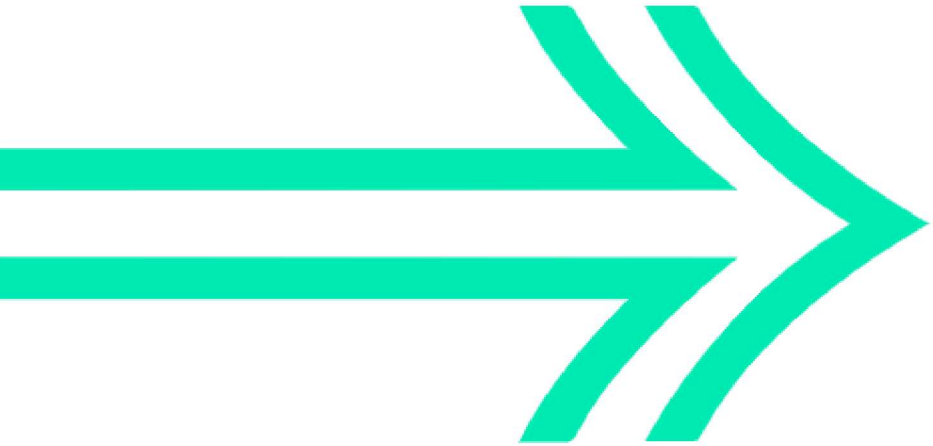
```
import React from 'react';
import {BrowserRouter, Routes, Route} from 'react-router-dom'
import Navbar from './components/Navbar';
import HomePage from './pages/HomePage';
import AboutPage from './pages/AboutPage';
import './App.css';
```



We set up App.js to import the required components for routing to take place.

React Router

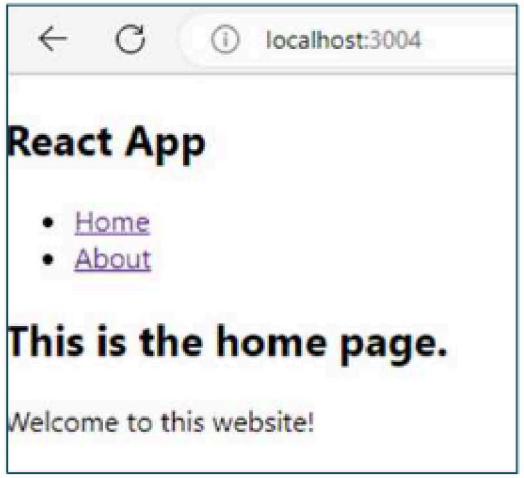
Finally, we build the structure and tell React which components to render when which link is clicked.

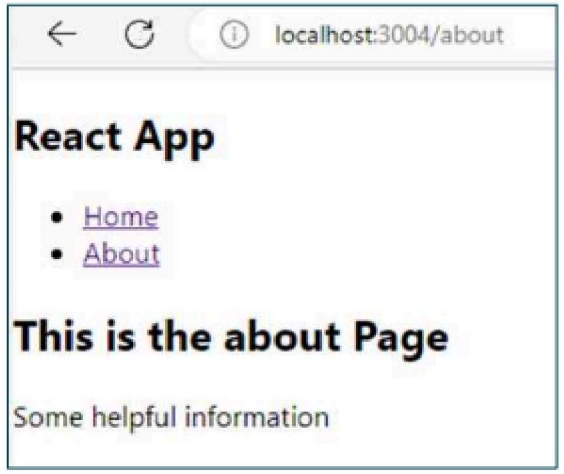


```
function App() {
  return (
  <BrowserRouter>
    <Navbar />
    <main>
      <Routes>
        <Route path="/" element={<HomePage />} />
        <Route path="/about" element={<AboutPage />} />
      </Routes>
    </main>
  </BrowserRouter>
  );
}

export default App;
```

React Router

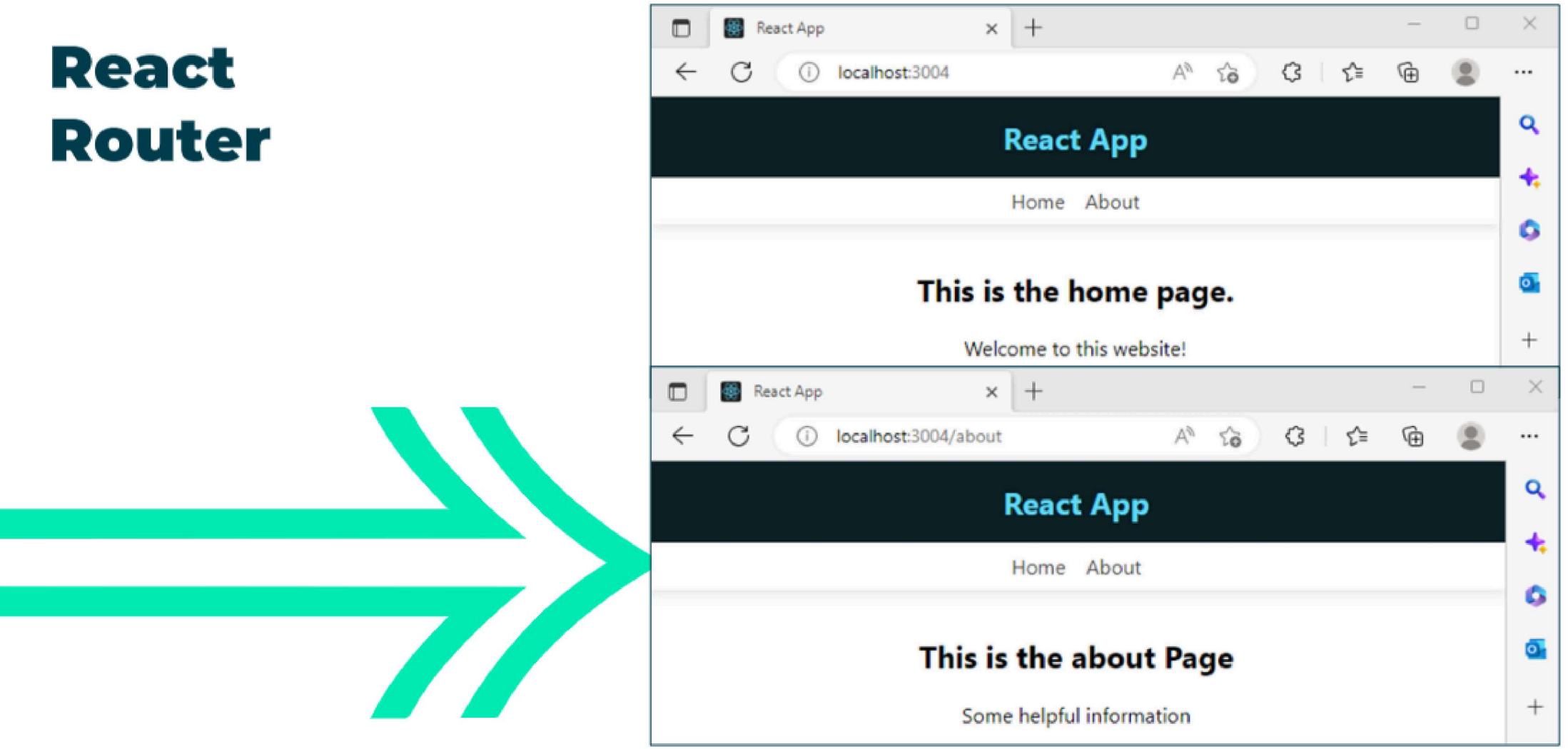
Finally, we build the structure and tell react which components to render when which link is clicked.

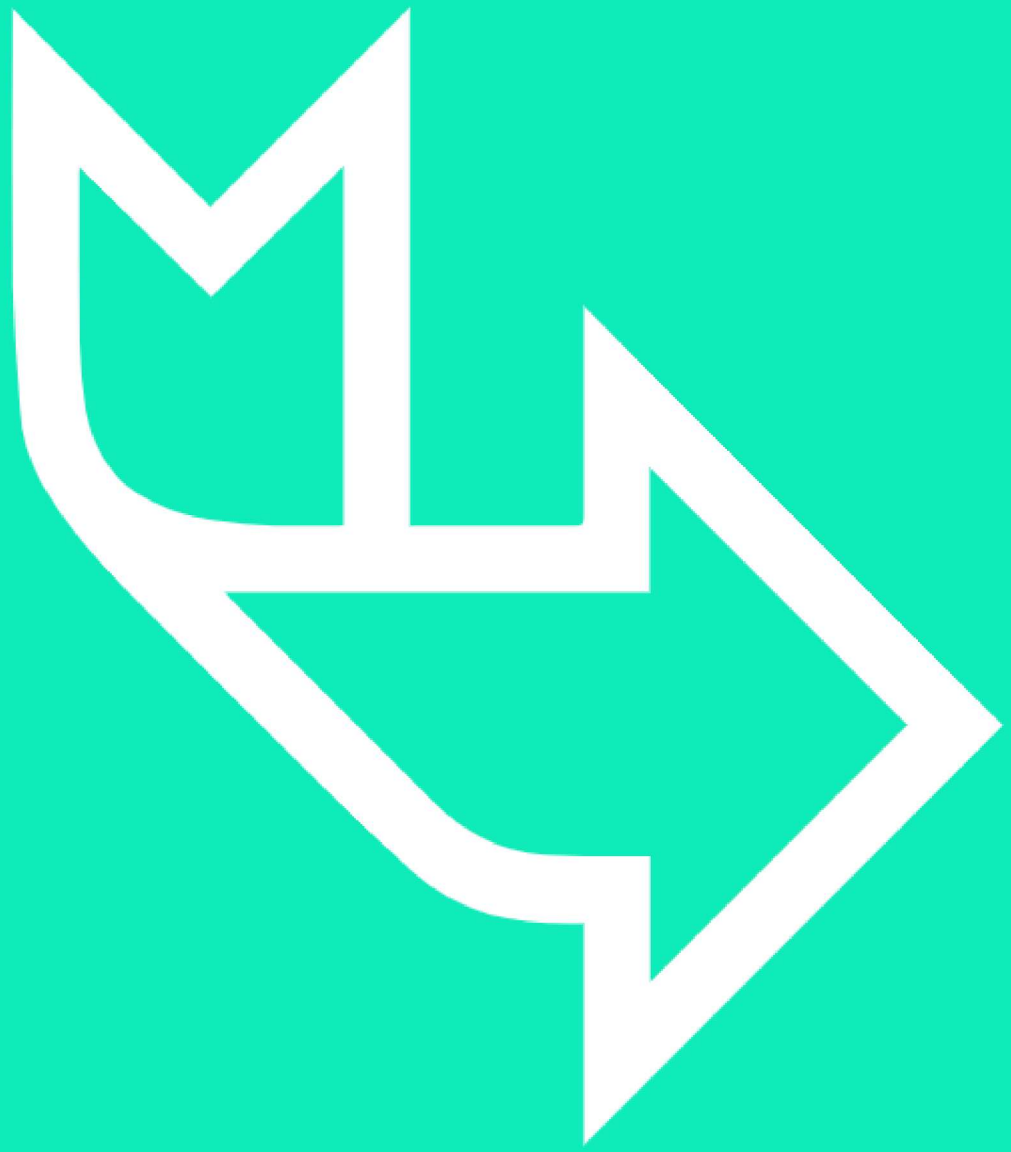




React Router

After some styling with CSS, the final output is:





QuickLab 6

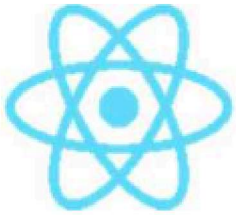
Creating Routes

HANDS-ON PROJECT

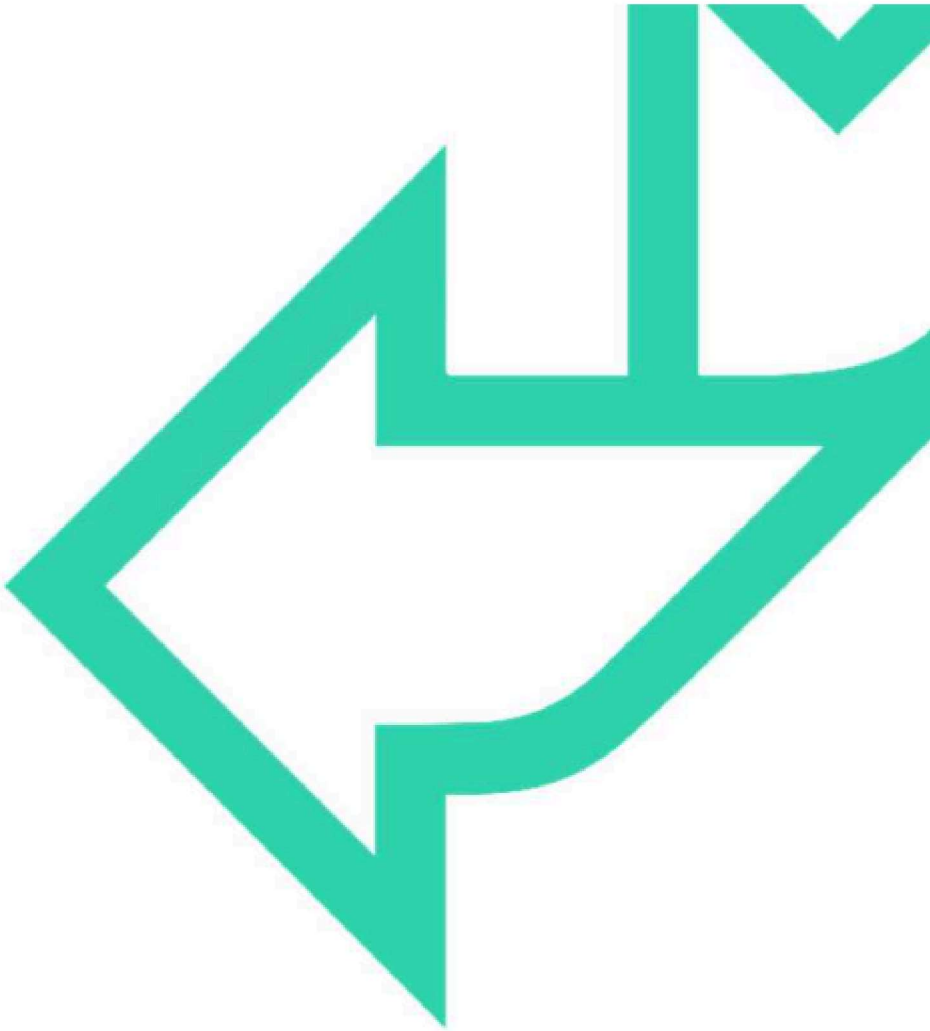
At your own pace, work through:

- Challenge 1

QA POWERING POTENTIAL

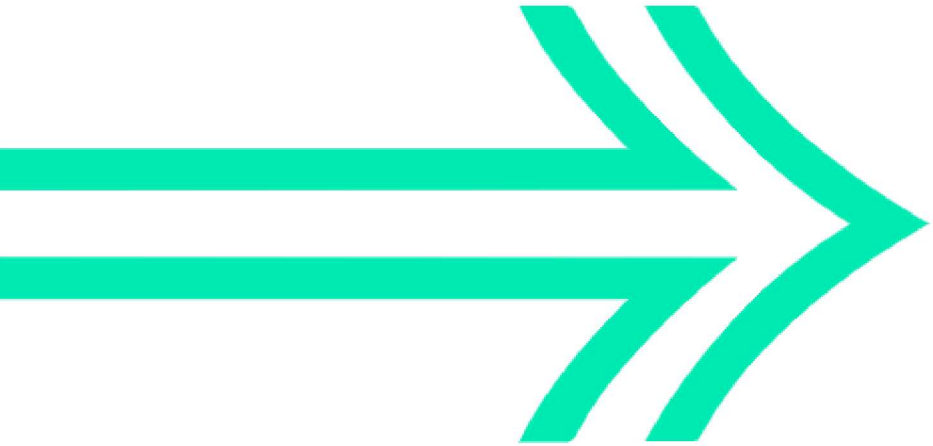


HANDS-ON WITH REACT



Handling events

In order to respond to user interactions like clicking, hovering, and focusing, you'll need to add **event handlers** to your elements.

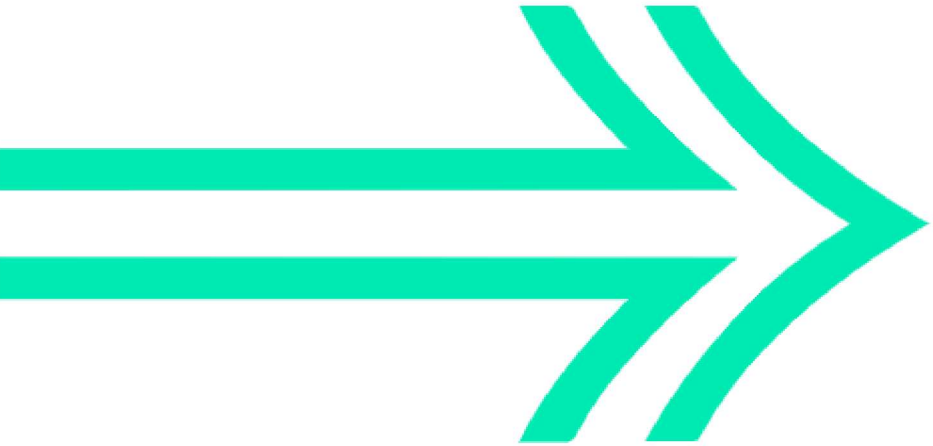


This is an extension of the learning completed in the JavaScript content and can be used extensively in JSX.

Handling events

These function identically in JS, but remember, in react any pure JavaScript must be put in curly braces, e.g.:

```
function App() {
  return (
    <div className="App">
      <h1>A simple Event</h1>
      <button onClick={() => {
        alert("Hi")
      }}>Click Me</button>
    </div>
  );
}

export default App;
```



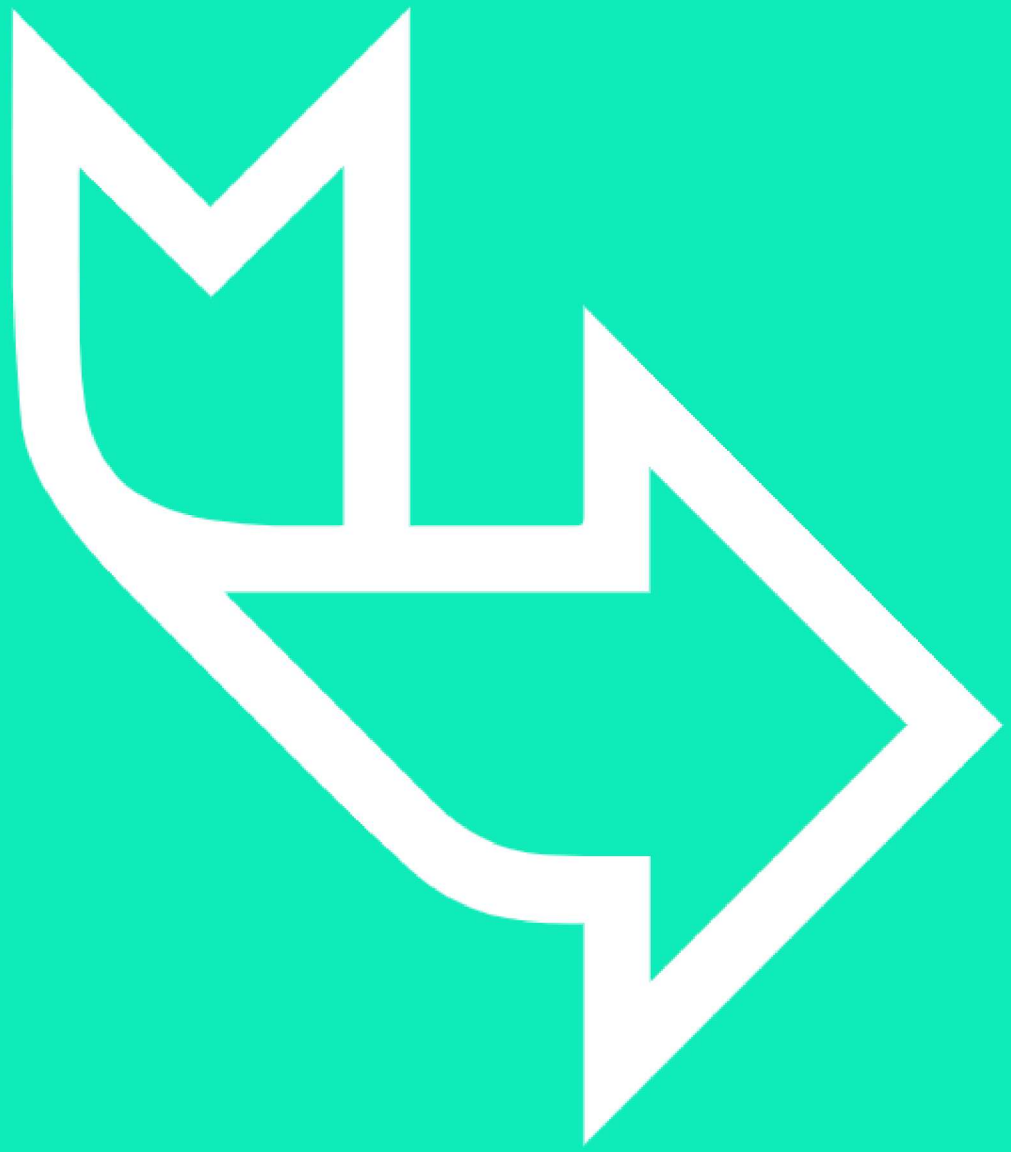
The anonymous function stops automatic processing of the script.

Handling events

We tend to put the outcome of these events inside a component and import when needed:

```
const Alert = () => {
    return (
        <>
            {alert("Hi")}
        </>
    );
}

export default Alert;
```

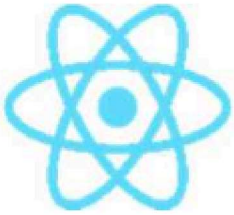
```
import Alert from './Alert';

function App() {
  return (
    <div className="App">
      <h1>A simple Event</h1>
      <button onClick={Alert}>Click Me</button>
    </div>
  );
}

export default App;
```

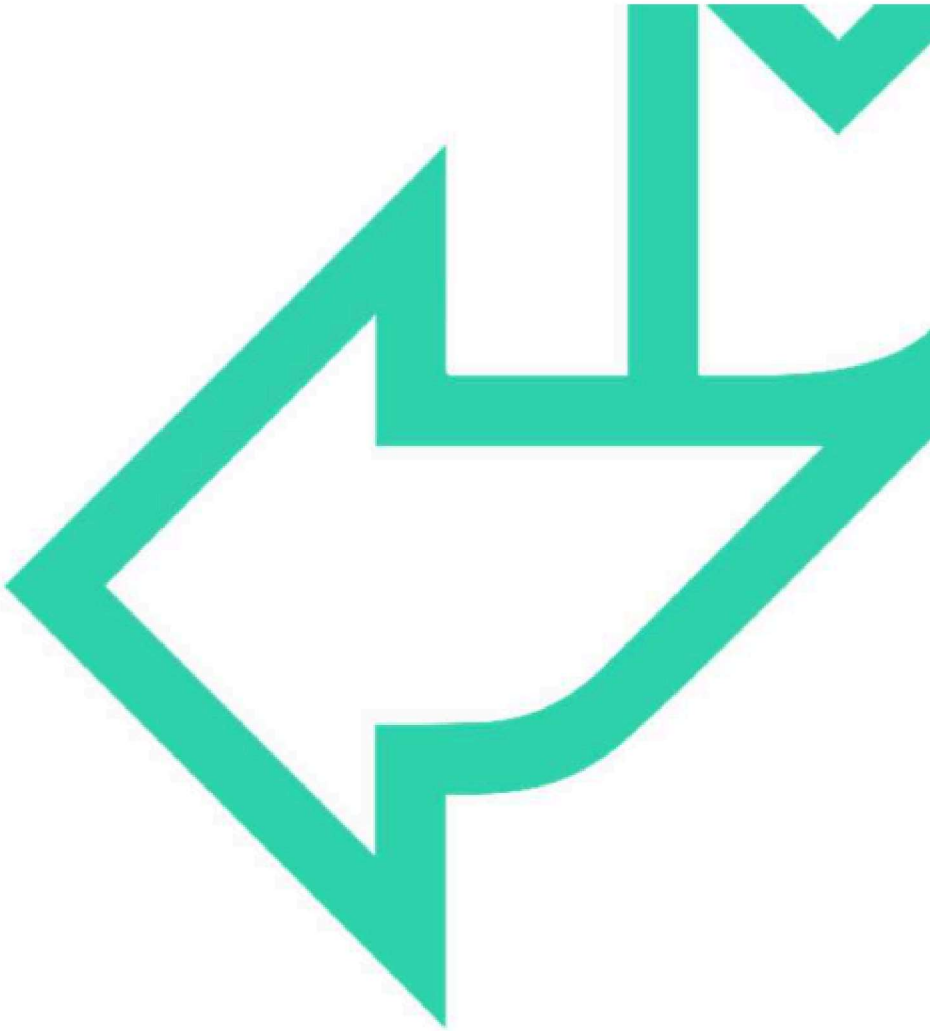


QuickLab 7

Event Handling

QA POWERING POTENTIAL





HANDS-ON WITH REACT

React state

Think of a **volume slider** component on a music player app. The volume can be changed over the lifetime of the app.

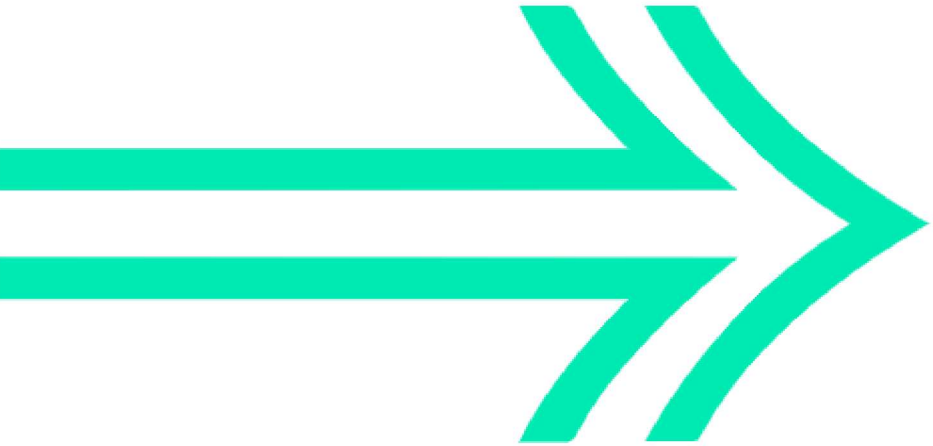
In React, **data that changes over time** is called **state**. State is like a component's memory.

In the example above, the volume would be stored as a **state value** so that it could be changed over time.

React state

State should be considered as variables in react. State is also considered to be the ‘single source of truth’.

Props should never be changed, but what we can do instead is deconstruct them into State. This is what is then displayed in the Virtual Dom.



To use state, all we need to do is import it at the top of our component and create the states (variables) within our functions.

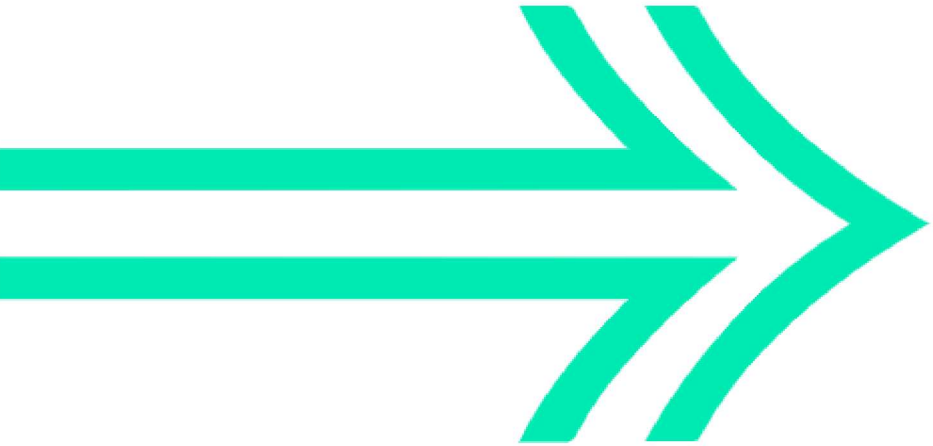
```
import { useState } from 'react'
```

React state

Creating State is now very easy in our code.

We simply create a new const and create an array made up of the State name (variable name) and a function which defines how to change the state.

Finally, we set the initial value of the State.



```
const [number, setNumber] = useState(3)
```

State name

Function to change State

Starting Value

React state

We can then output the value of the State by simply calling the name in the code, just like a JavaScript variable.

```
<h1>The number is</h1>
<h2>{ number }</h2>
```

The number is

3

Later in the code, I can create a button which calls the setNumber() function to update the state. This edits the DOM to reflect the new value of State.

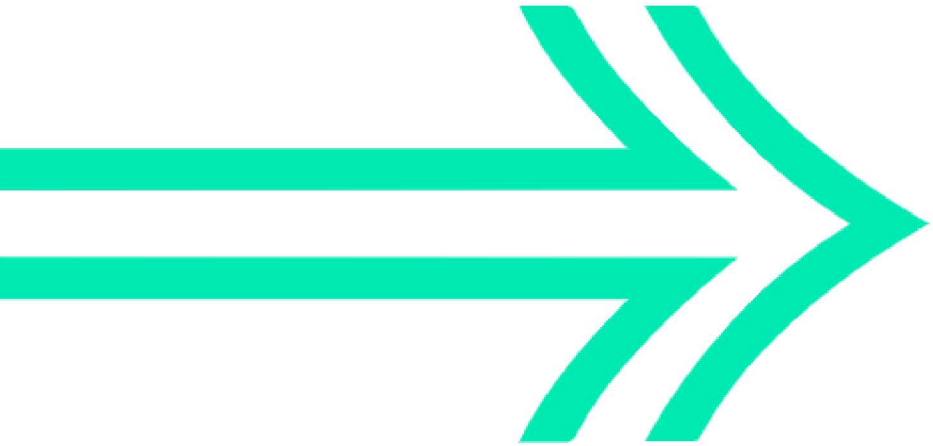
```
<button onClick={() => (
  setNumber(5)
)}>Change State</button>
```

The number is

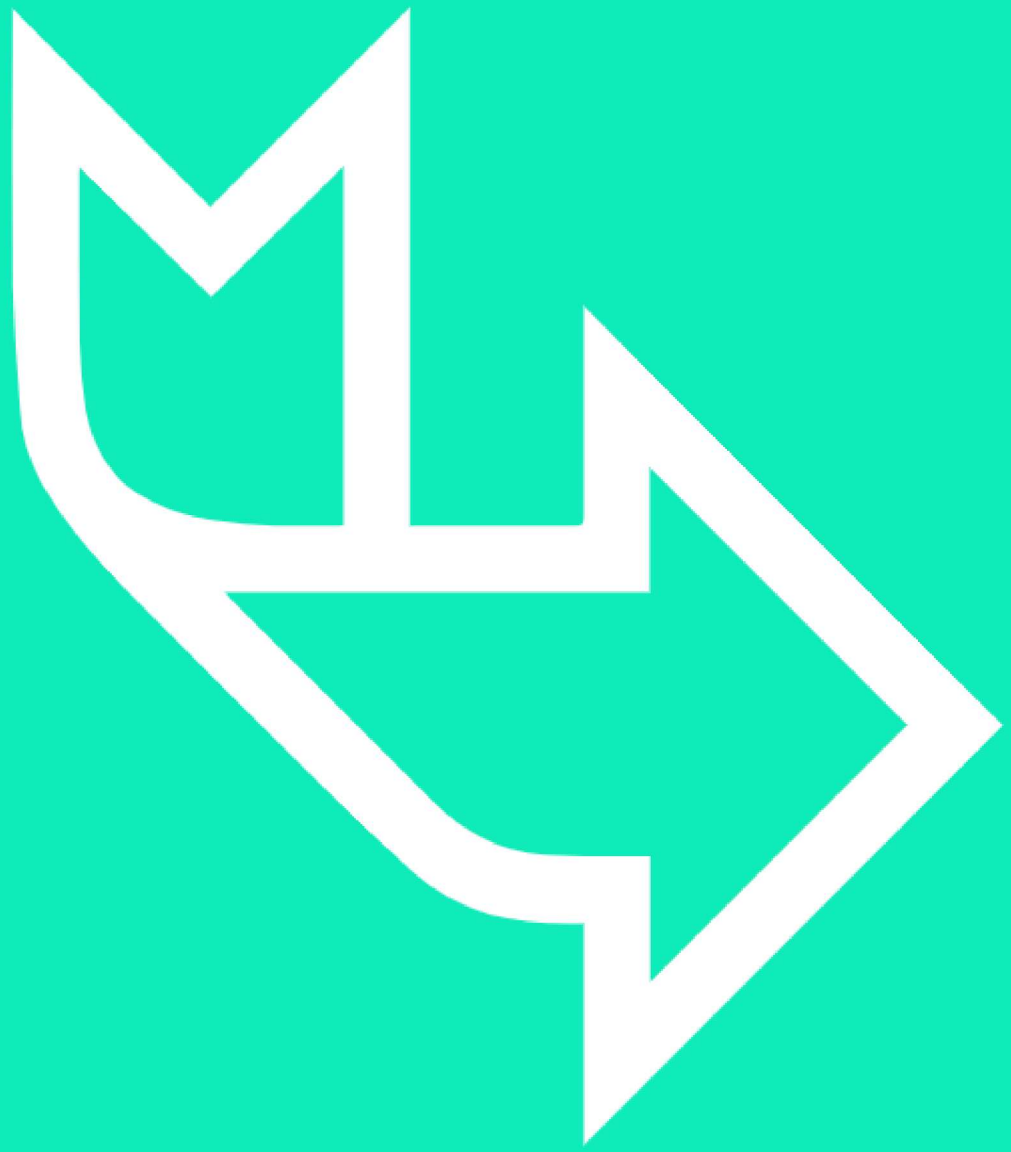
5

React state

The Process is the same for all State creation and updating:



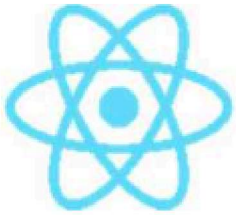
```
function App() {

  const [name, setName] = useState("Dave")
  const [number, setNumber] = useState(3)
  return (
    <div className="App">

      <h1>The number is</h1>
      <h2>{ number } {name}</h2>

      <button onClick={() => (
        setNumber(5), setName("Frank")
      )}>Change State</button>
    </div>
  );
}

export default App;
```



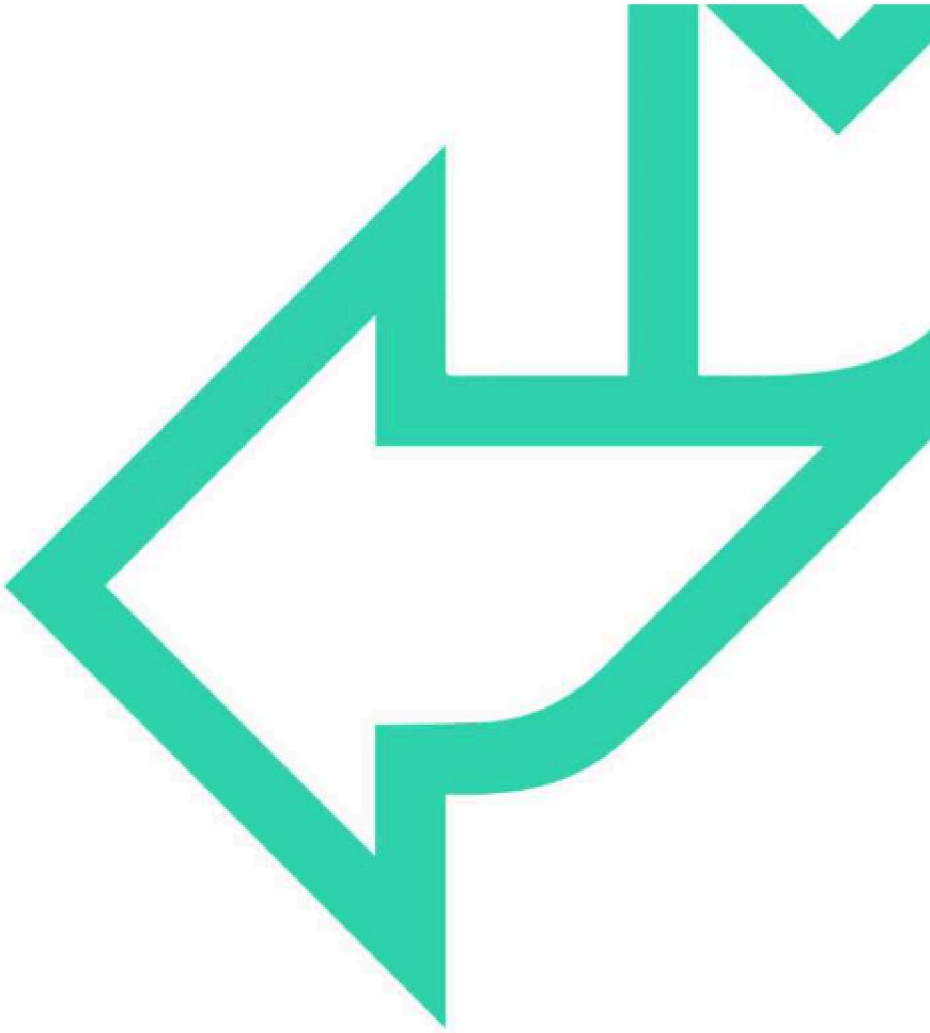
QuickLab 8

State

QA POWERING POTENTIAL



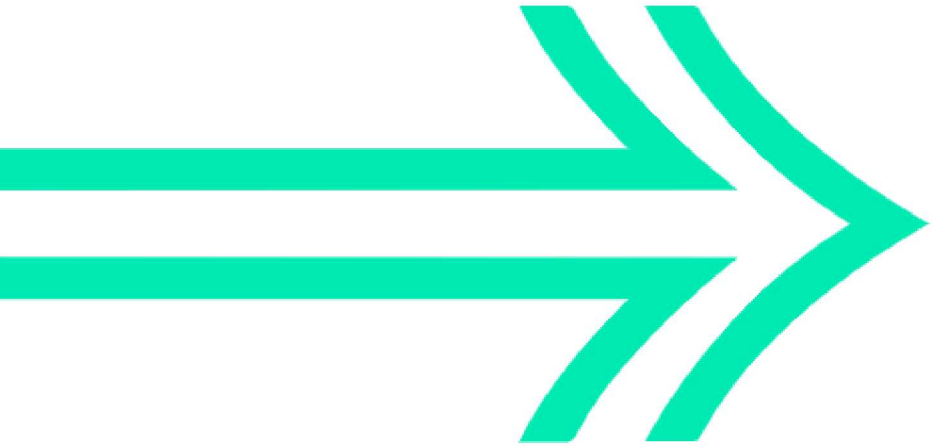
HANDS-ON WITH REACT



Inverse data flow

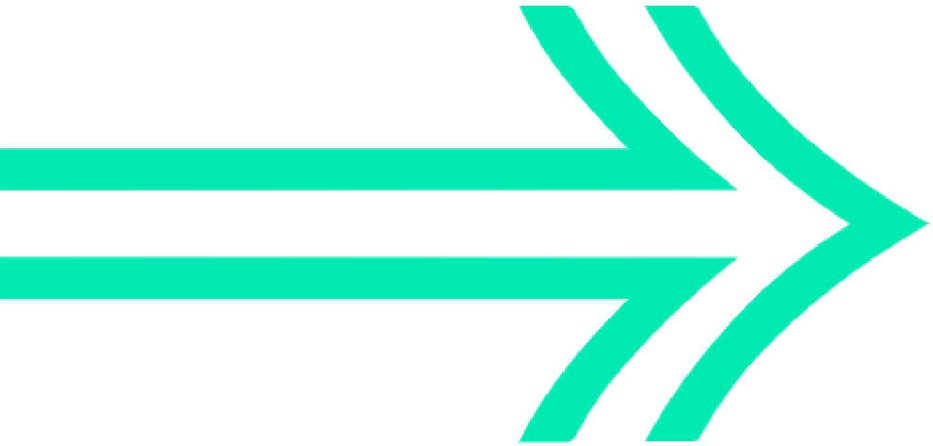
A component that contains another component is called a “**parent**”, and the contained component is called a “**child**”.

In React, **data flows down from parent to child**, not the other way around.



However, sometimes child components need to **trigger state changes higher up** in the hierarchy. This is possible, but we’ll need to think a little differently.

Inverse data flow

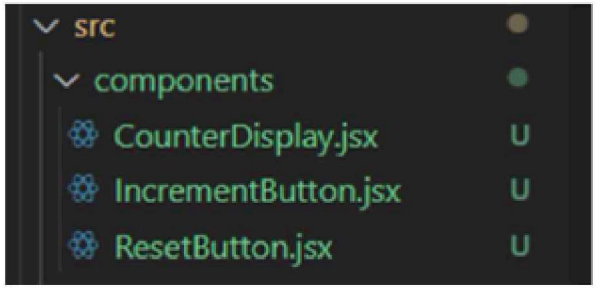


We sometimes need to pass State back up into the parents, just as we do props. This means we can change the State in the child and then pass it up to allow it to come back down.

This may seem counter intuitive, but it is important to consider that there may be multiple components and children that all depend on this State, we have to change it for all of them at the same time.

State is the ‘single source of truth’.

Inverse data flow



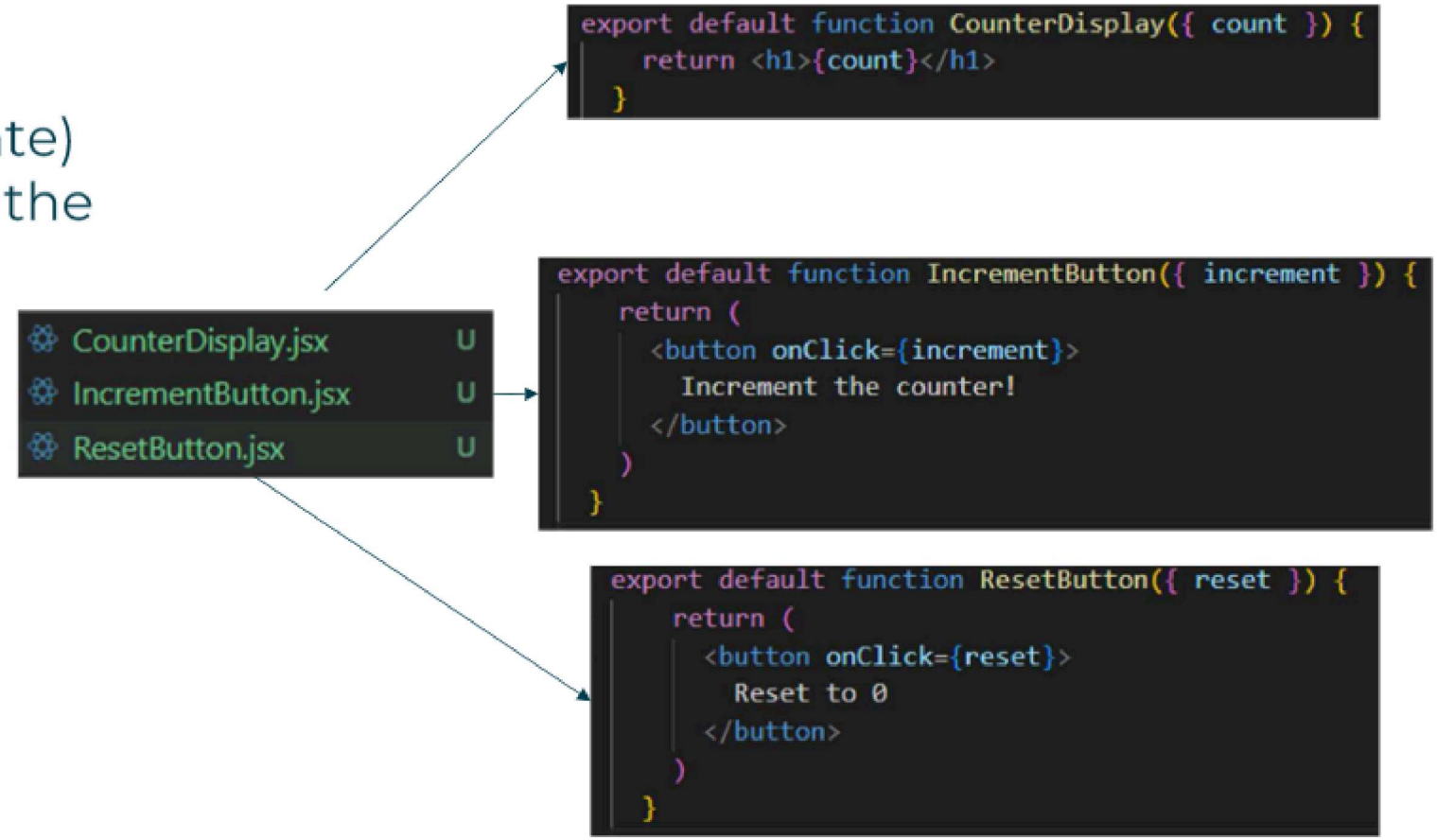

```
const [count, setCount] = useState(0)

function handleIncrement() {
  setCount((previous) => previous + 1)
}

function handleReset() {
  setCount(0)
}
```

Here you can see that we have created three components. All of them are called into **App.js** and all of them pass. There are also three functions which are passed as Props to the parents.

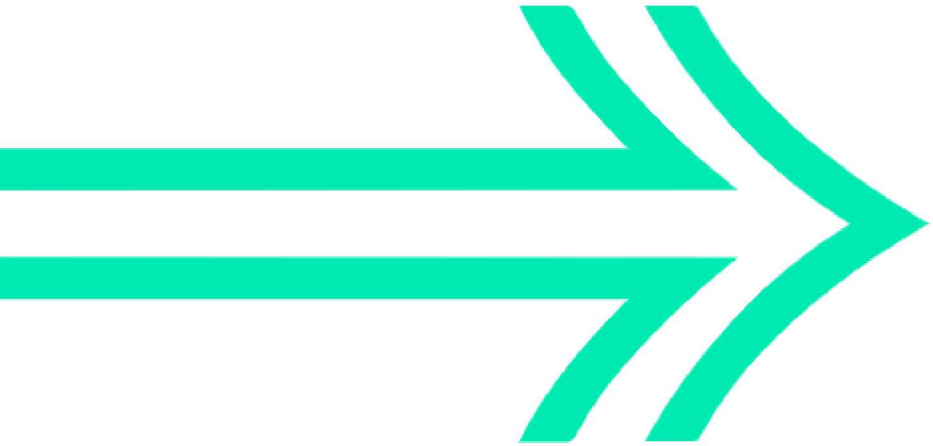
```
return (
  <main>
    <CounterDisplay count={count} />
    <IncrementButton increment={handleIncrement} />  
    <ResetButton reset={handleReset} />
  </main>
```

Inverse data flow

Each component takes the props (which is really State) and then updates the Virtual DOM with that data.

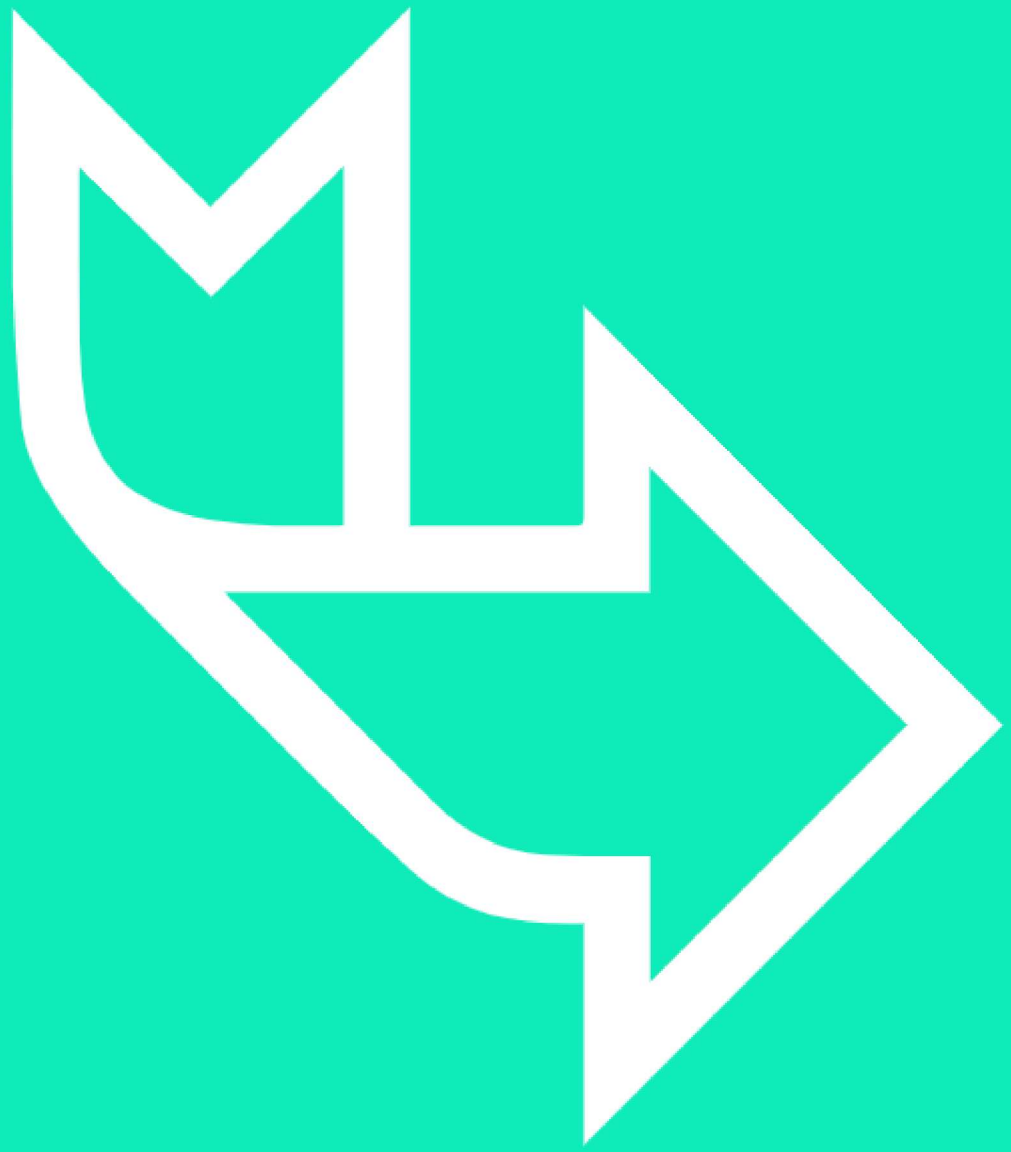


Inverse data flow



The result is that State created in the child is updated, passed to the parent as Props, and then returned to the Child.

This results in updating any and all children who are dependent on that state.



QuickLab 9

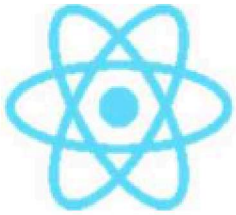
Inverse DataFlow

HANDS-ON PROJECT

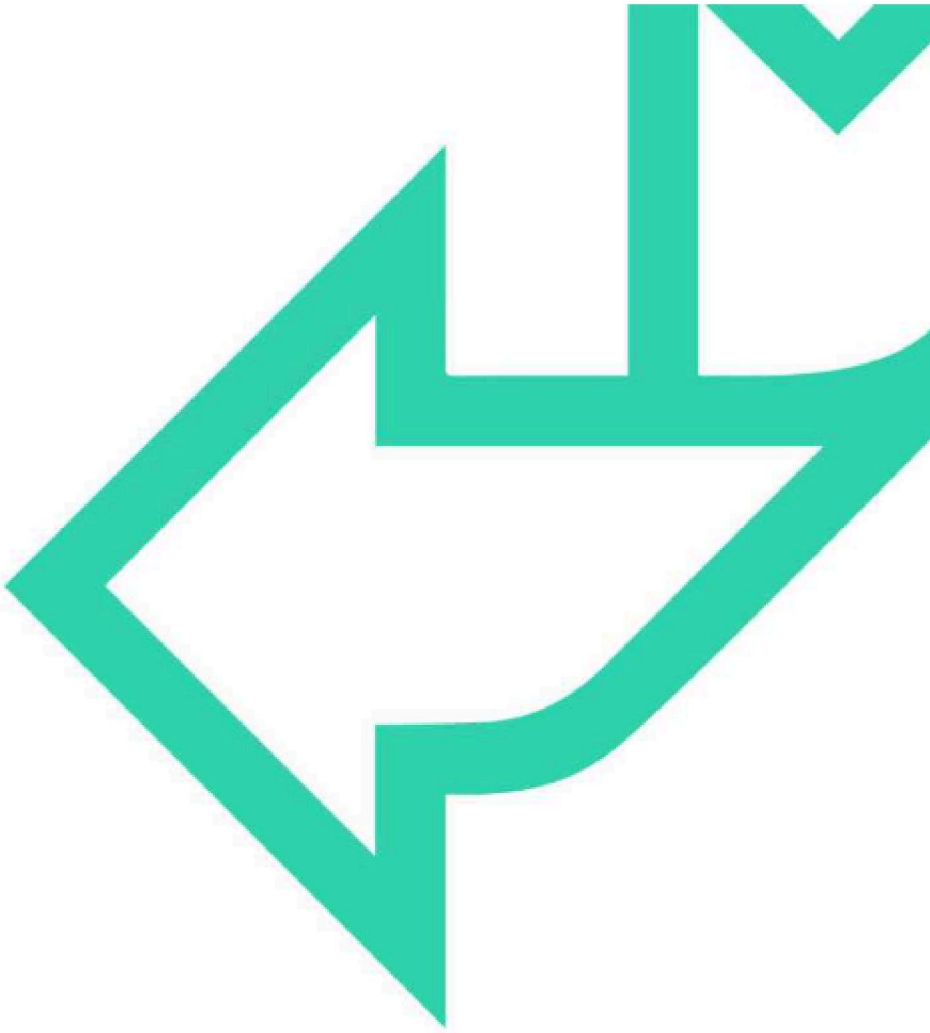
At your own pace, work through:

- Challenge 1
- Challenge 2

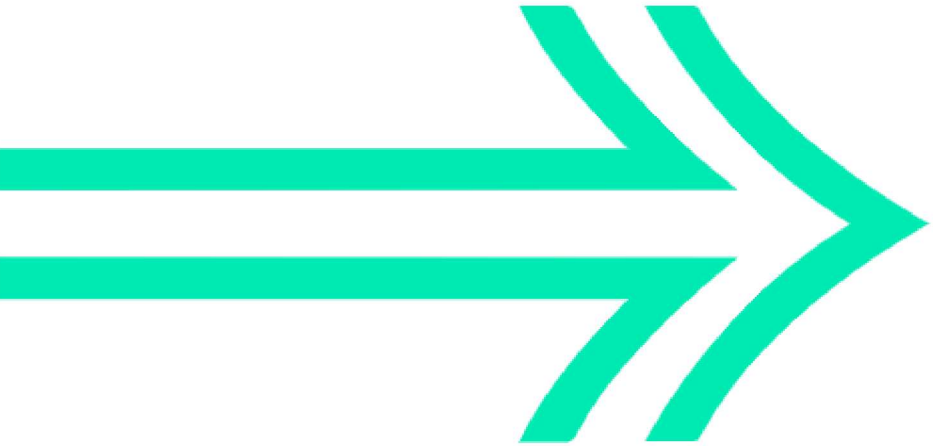
QA POWERING POTENTIAL



HANDS-ON WITH REACT



The useEffect hook



Sometimes, a React component needs to interact with the 'outside world', performing actions that are not directly related to the UI output of a component. For example:

- making a request to an external API.
- interacting with browser APIs (e.g., writing to local storage).

These actions are called "**side effects**". The **useEffect** hook allows us to manage side effects without interfering with the rendering of the component.

Before we see a practical example, let's cover the different ways in which useEffect works.

How useEffect works (1/2)

```
function Component() {

  useEffect(() => {
    // Side effect
  })

}
```

Takes in a function containing your side effect code

This will run the side effect **after every render** (the first time the component displays, and every time it updates).

```
function Component() {
  const [state, setState] = useState( ... )

  useEffect(() => {
    // Side effect
  }, [state])

  ...
}
```

This will run **after the first render** and after any time the **state value changes.**

```
function Component({ prop1, prop2 }) {

  useEffect(() => {
    // Side effect
  }, [prop1])

  ...
}
```

Second argument is the "dependency array"

This will run the side effect **after the first render** and after any time the **value of prop1 changes.**

```
function Component({ prop1, prop2 }) {

  useEffect(() => {
    // Side effect
  }, [prop1, prop2])

  ...
}
```

Multiple dependencies

This will run **after the first render** and after any time **any of the dependencies change** (as long as the dependency is a prop or state value).

How useEffect works (2/2)

```
function Component() {

  useEffect(() => {
    // Side effect
  }, [])

}
```

Empty array – what will happen?

This will run **after first render only** (when the component **mounts**).

```
function Component() {

  useEffect(() => {
    // Side effect

    return () => {
      // Side effects cleanup function
    }
  }, [])

  ...
}
```

If you need to **clean up any side effects**, return a function that handles the cleanup.

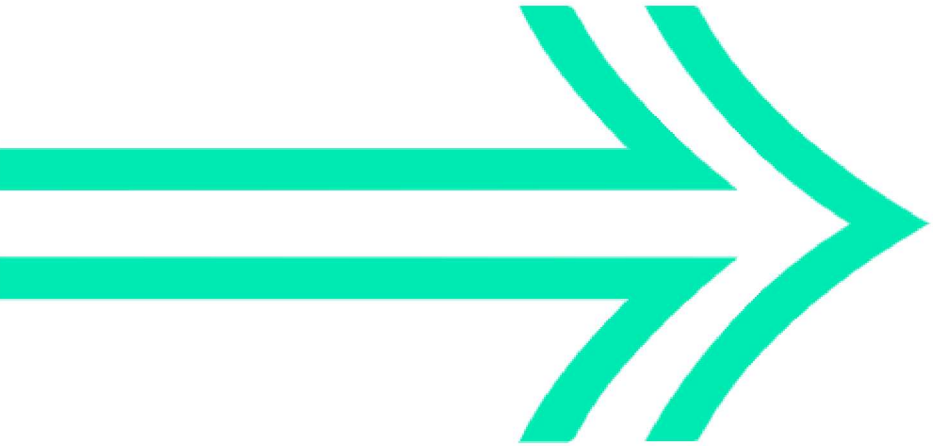
Pitfall: forgetting the dependency array

Remember, with no dependency array, this will run on **every render**. If you are doing anything that uses up a quota or costs money (e.g., reads from a database), having no dependency array is likely a mistake which will have consequences if your component re-renders often.

Even worse, if you modify state within this effect, then the state change will trigger a re-render, which will trigger the effect, which will trigger a re-render, which will trigger the effect... An infinite loop. This will also affect the performance of your app.

```
function Component() {

  useEffect(() => {
    // Side effect
  })

  ...
}
```

The useEffect hook



Now that we've looked at how useEffect works, let's see it in practice.

Remember, useEffect needs to be imported like useState and by default, runs when the page first renders.

We change when we want the code to run but adding dependencies in the useEffect call.

We will use State set as Boolean true or false to log something to the console each time another state is changed.

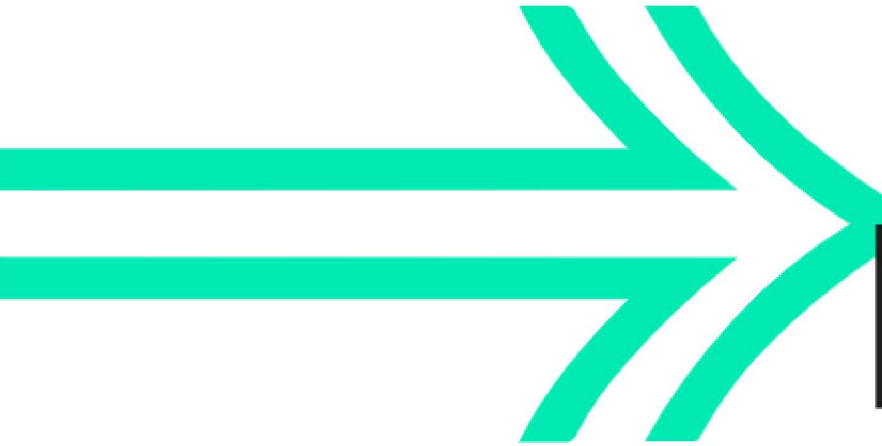
The useEffect hook

He we have two States, one number, the other Boolean.

```
const [count, setCount] = useState(0)
const [isLoggingOn, setIsLoggingOn] = useState(true)
```

A button exists to toggle logging.

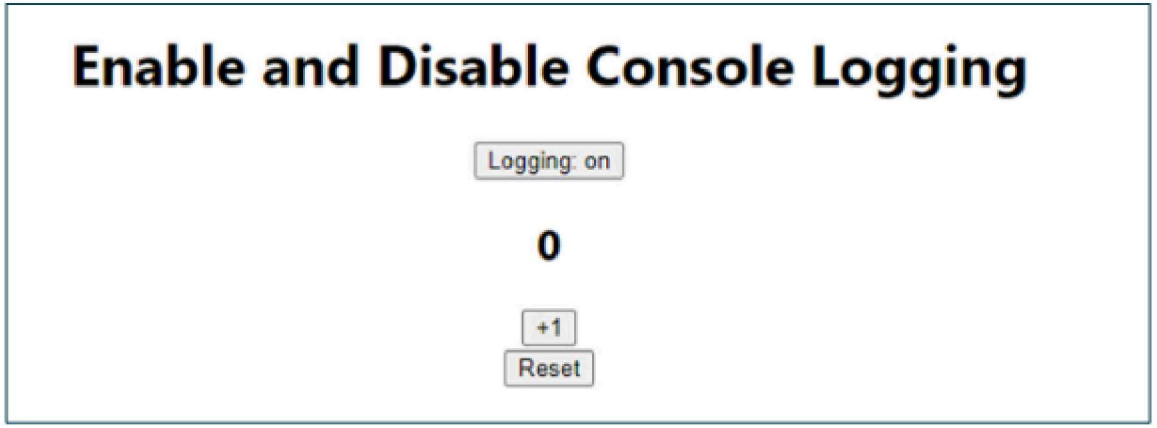
```
<button onClick={() => setIsLoggingOn((prev) => !prev)}>
  Logging: {isLoggingOn ? 'on': 'off'}
</button>
```

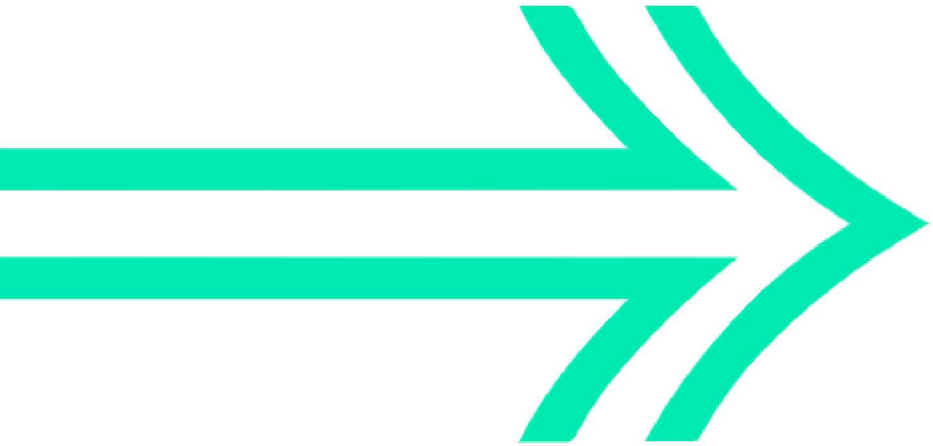


The count is then displayed, along with a button to increment and another to reset.

```
<h2>{ count }</h2>

<button onClick={() => setCount((previous) => previous +1)}>+1</button><br />
<button onClick={() => setCount(0)} >Reset</button>
```

The useEffect hook



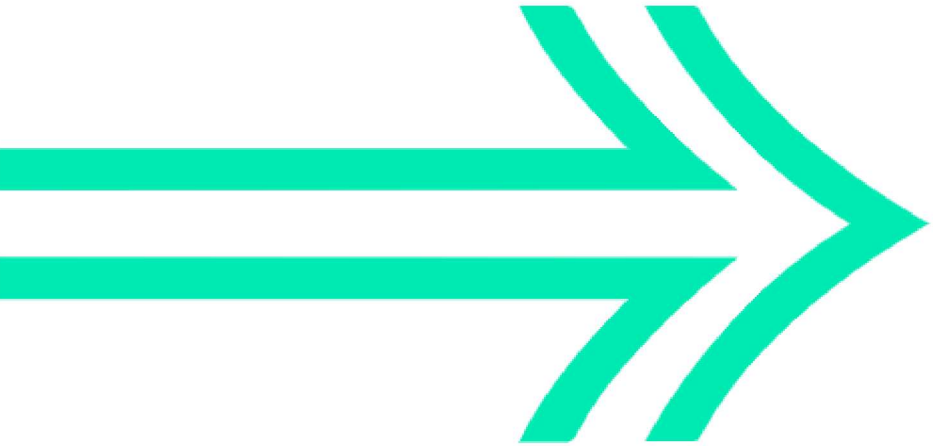


The useEffect hook

Let's add a useEffect to control when the logging happens.

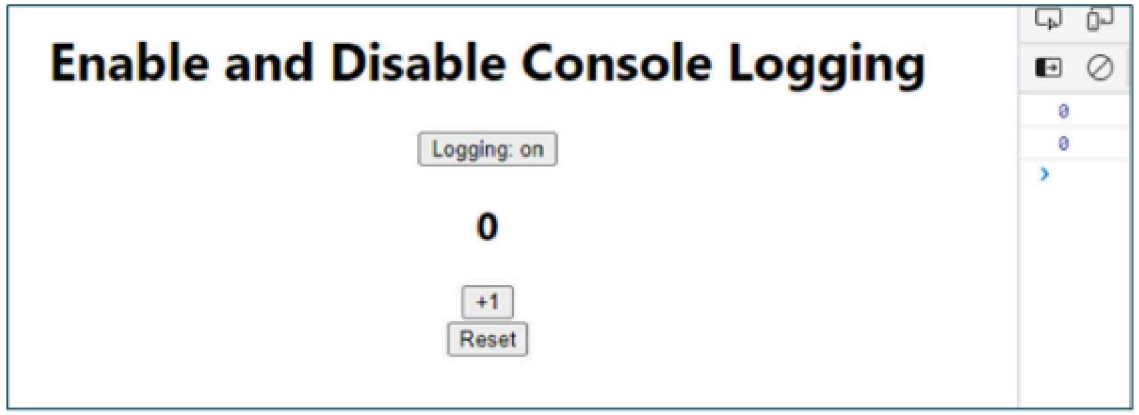
Each time the count State changes, this useEffect runs. It checks to see if logging is true or false.

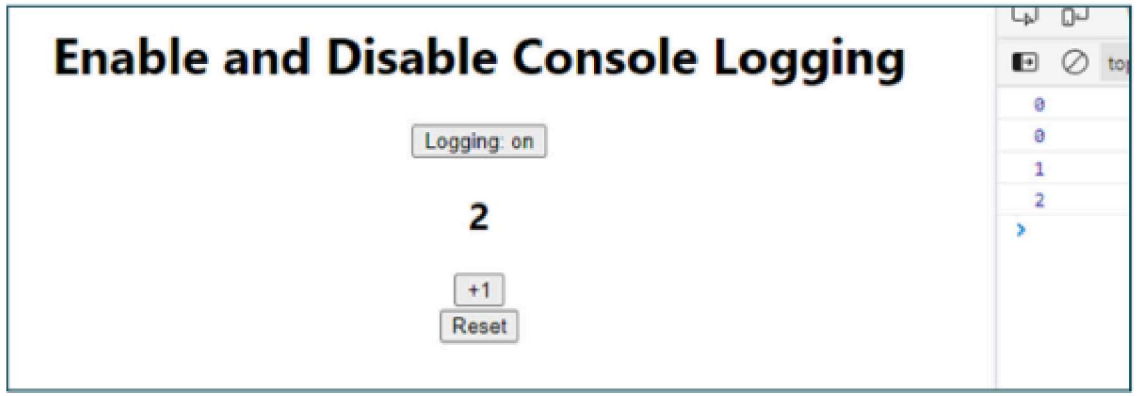
If true, it logs the value of count to the console.



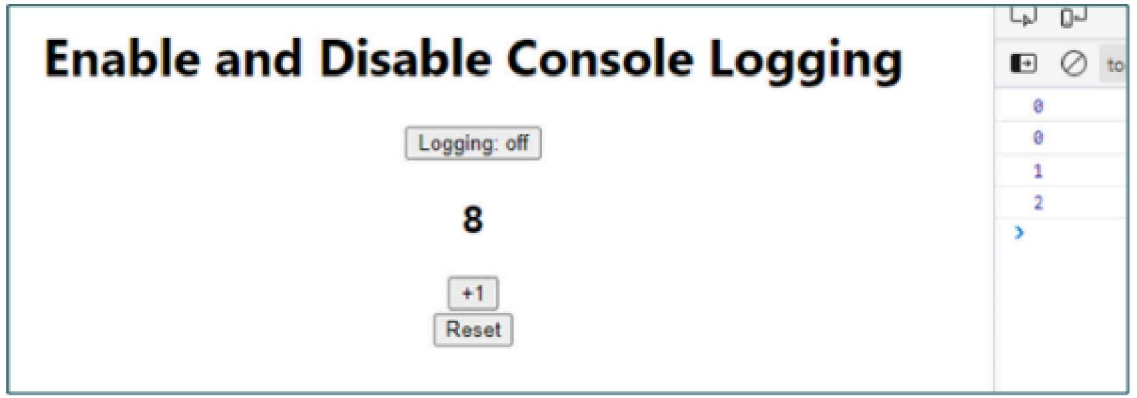
```
useEffect(() => {
  if (isLoggingOn) {
    console.log(count)
  }
}, [count])
```

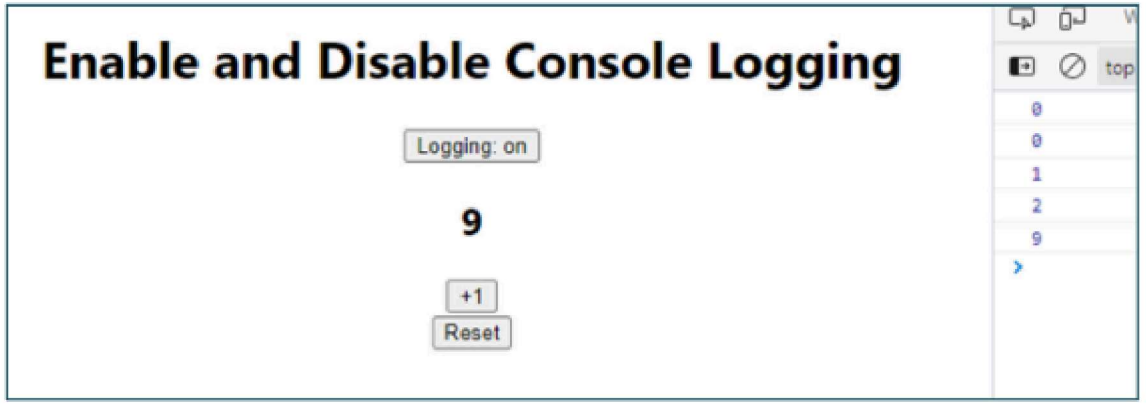
The useEffect hook



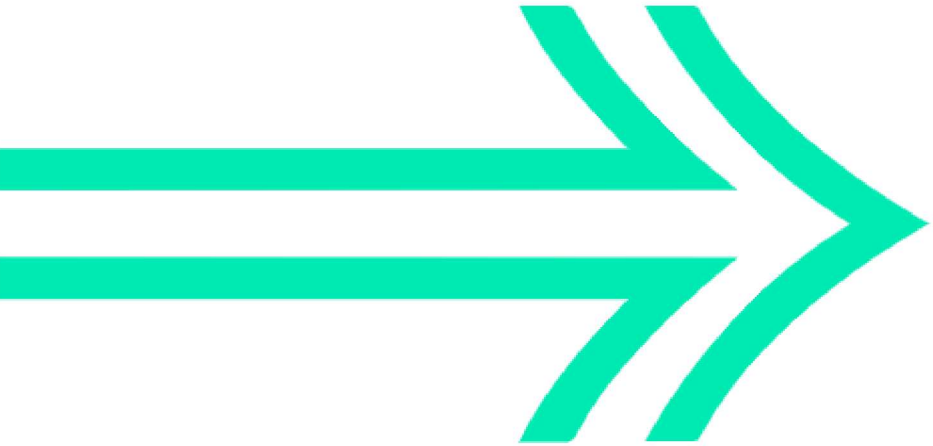


The useEffect hook

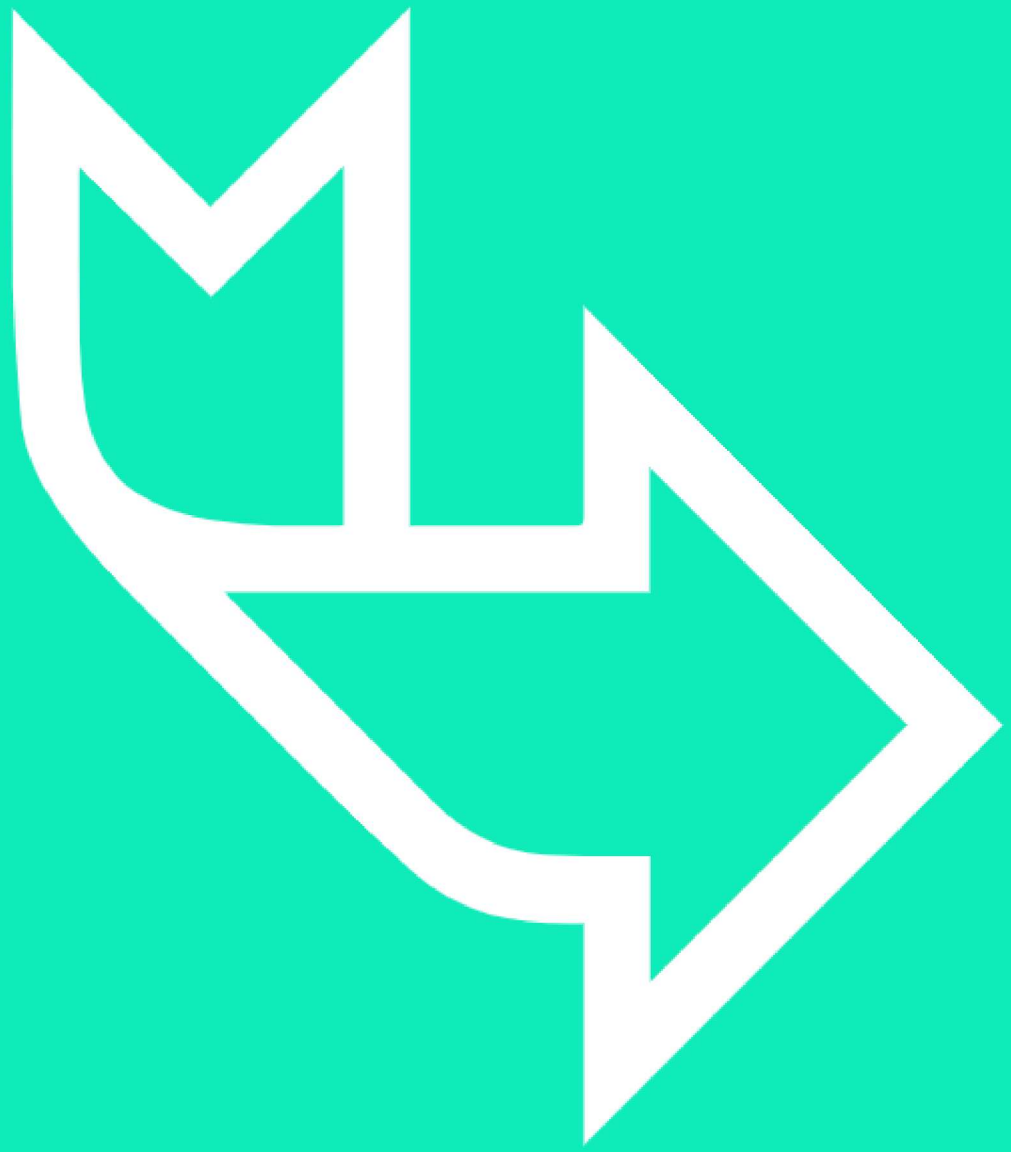




The useEffect hook



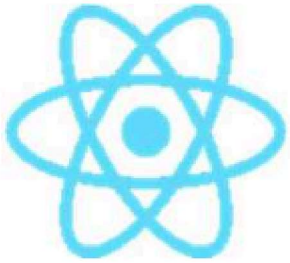
```jsx
import {useState, useEffect} from 'react'
import './App.css';

function App() {
  const [count, setCount] = useState(0)
  const [isLoggingOn, setIsLoggingOn] = useState(true)

  useEffect(() => {
    if (isLoggingOn) {
      console.log(count)
    }
  }, [count])


  return (
    <div className="App">
      <h1>Enable and Disable Console Logging</h1>

      <button onClick={() => setIsLoggingOn((prev) => !prev)}>
        Logging: {isLoggingOn ? 'on': 'off'}
      </button>

      <h2>{ count }</h2>

      <button onClick={() => setCount((previous) => previous +1)}>+1</button><br />
      <button onClick={() => setCount(0)} >Reset</button>

    </div>
  );
}

export default App;
```

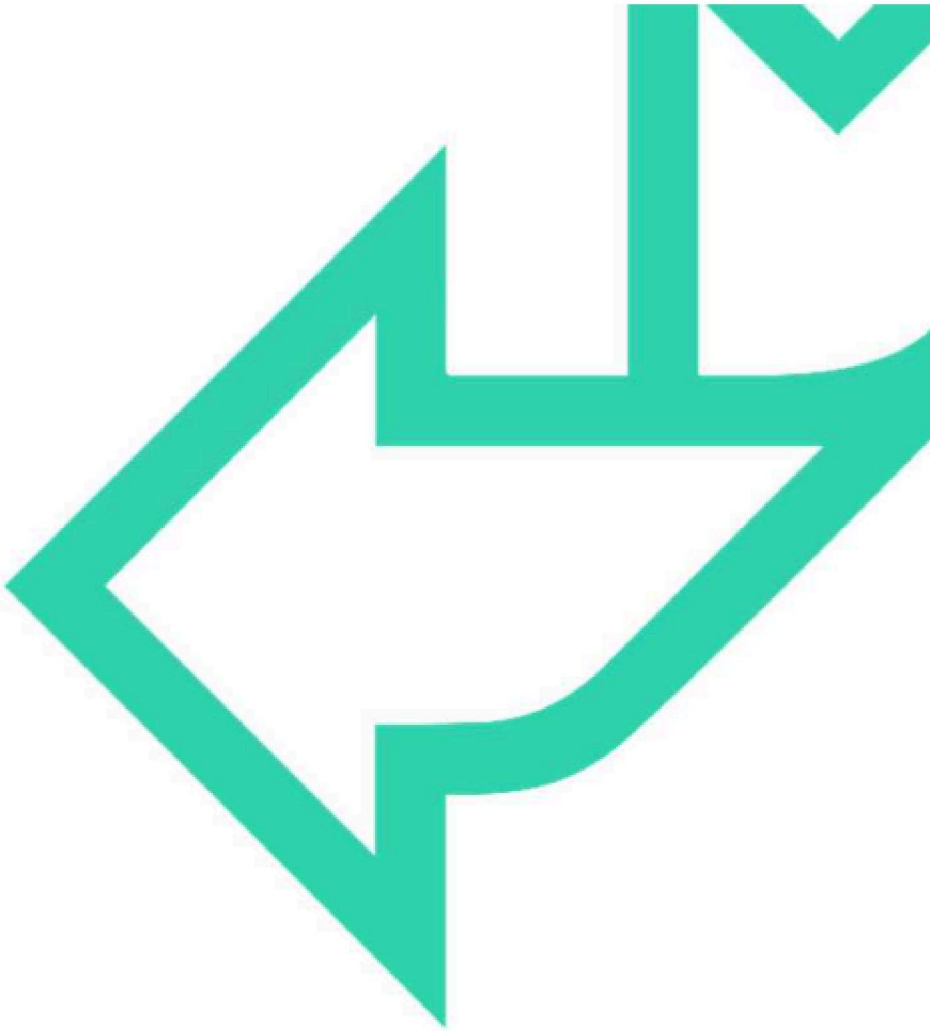


QuickLab 10

useEffect

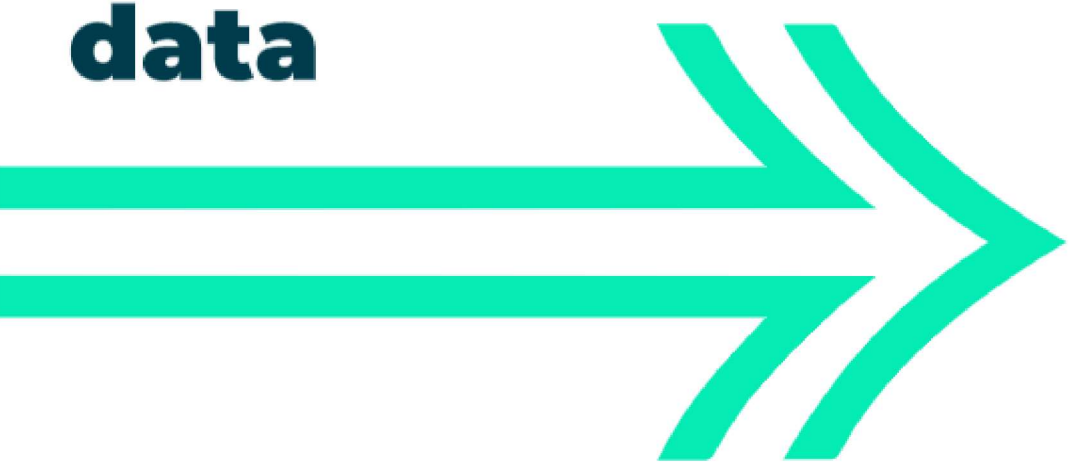
QA POWERING POTENTIAL





HANDS-ON WITH REACT

Using external data

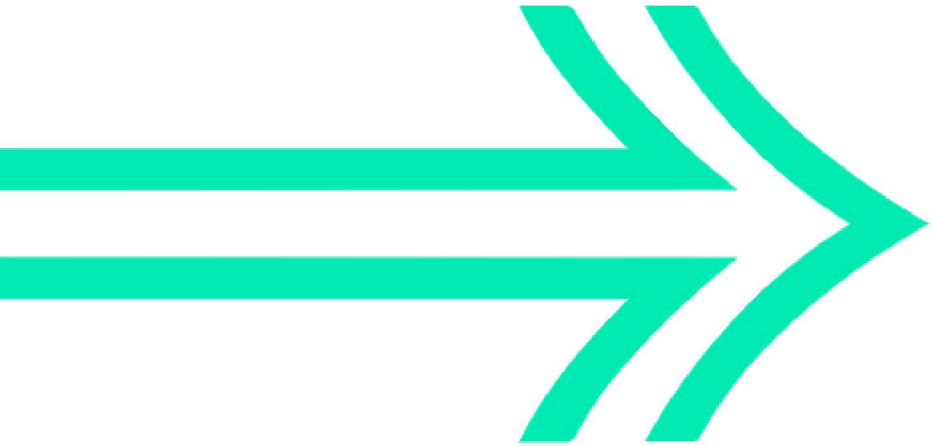


Sometimes your application will **rely on external data** – for example, you might need to use the result of an API call.

Let's look at how to achieve this in React.

This will be a very simplified version of a challenge later in the section, but this will introduce you to the fetch API in React.

Using external data



In this example, we will create some State and mock up fetching data to update that state. This example will be a simple image that we will pull from the internet

We set a const as a url:

```
const apiUrl = 'https://dog.ceo/api/breeds/image/random'
```

We create empty State:

```
function App() {
  const [imageUrl, setImageUrl] = useState('')
```

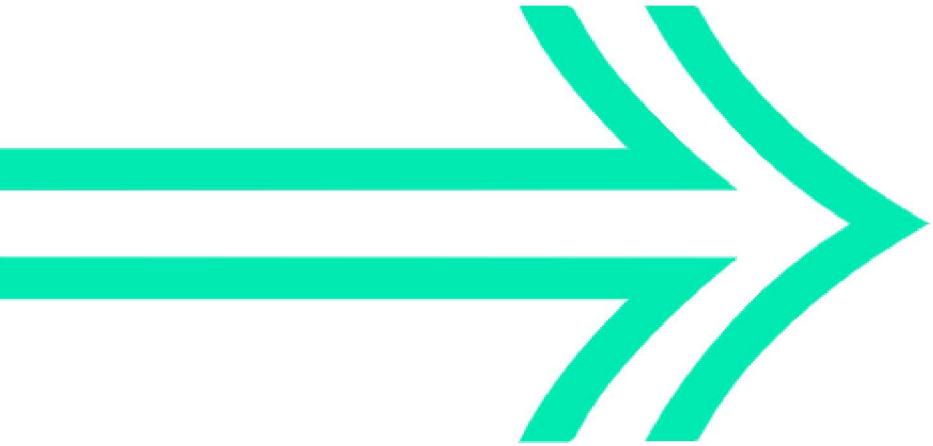
In the return, we set the image src to be the value of imageUrl:

```
return (
  <div className="App">
    <main>
      <h1>Go Fetch</h1>
      <img width={300} src={imageUrl} alt=""/>
    </main>
  </div>
);
}
```

Using external data

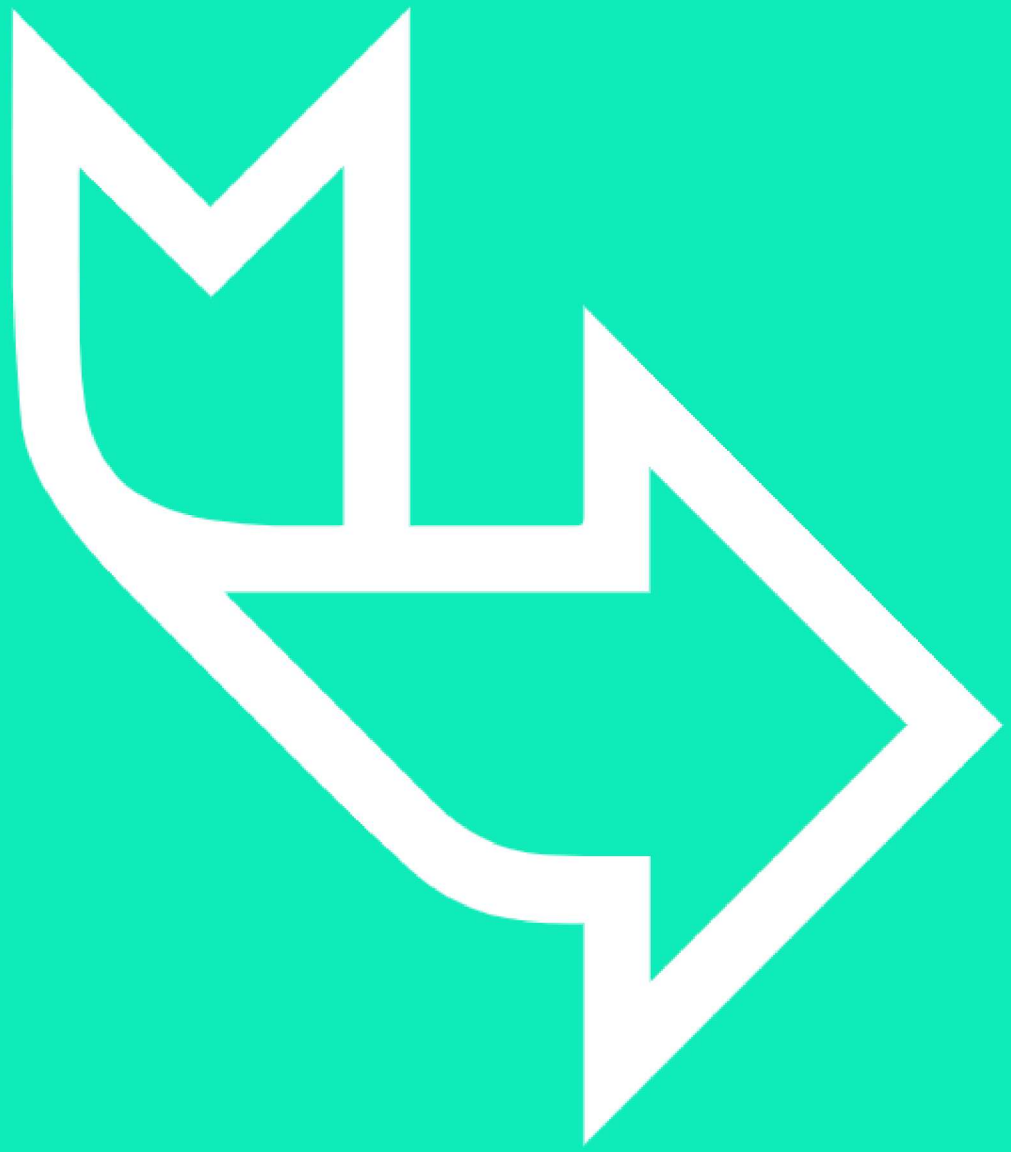
All we need to do now is a useEffect and a promise (fetch) to drag down the image and display it.

```
useEffect(() => {
  fetch(apiUrl)
    .then((response) => response.json())
    .then((data) => setImageUrl(data.message))
}, [] )
```



Notice, this only runs once when the page is initially rendered.

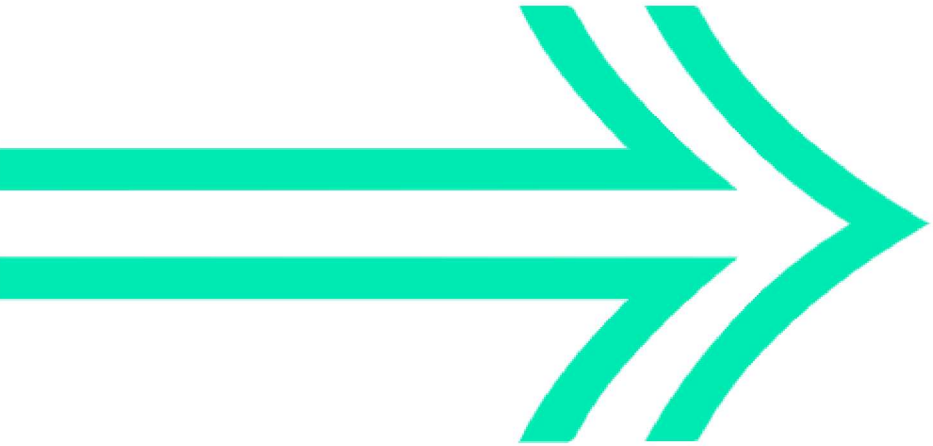
This means each time we refresh the page it collects the image again, which happens to be random.



QuickLab 11

useEffect to collect data

Using JSON Server

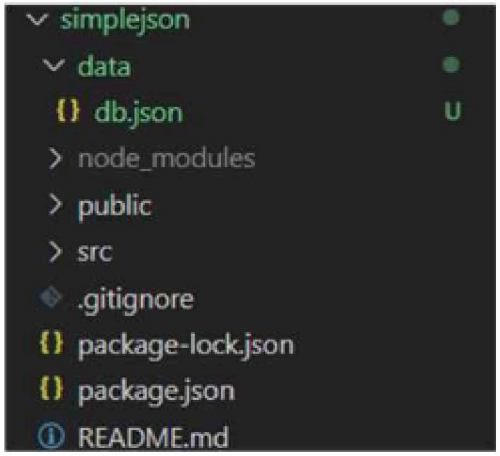


We can also use a Json server, running on our own machines to serve up JSON files.

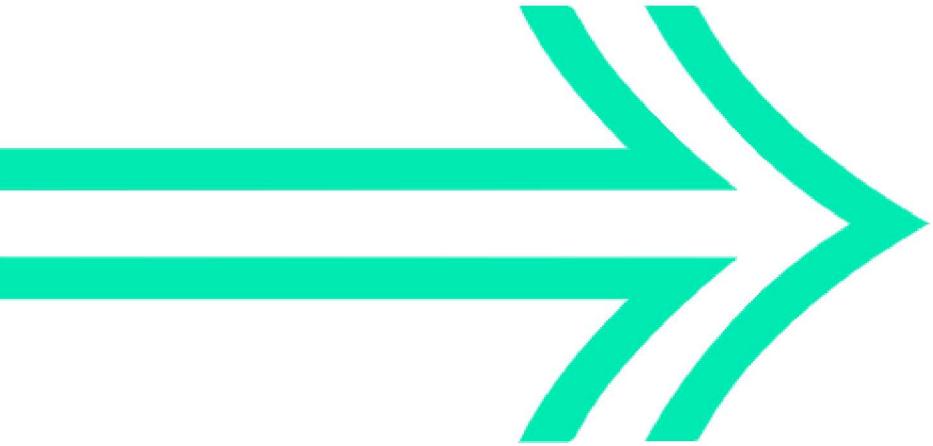
There is a huge advantage to this, which is that we can add and delete records to the file this way, meaning we can now enact permanent data change.

Using JSON Server

We can use NPX to install the Json server. This is similar to NPM, but pulls in instructions off the web, rather than storing them locally.



Using JSON Server



Because we need to run the Json server separately, we need to start a new terminal window in VSCode.

Once done, we need to type the following command:

```
npx json-server --watch data/db.json --port 8000
```

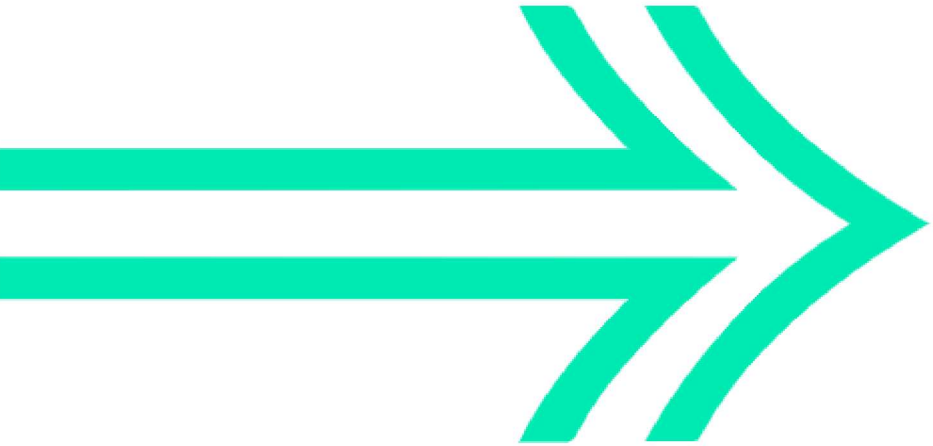
--watch allows us to monitor the contents.

--port 8000 changes the default (3000) as the react dev server is already running on that one.

Data/db.json is the path to the data.

Using JSON Server

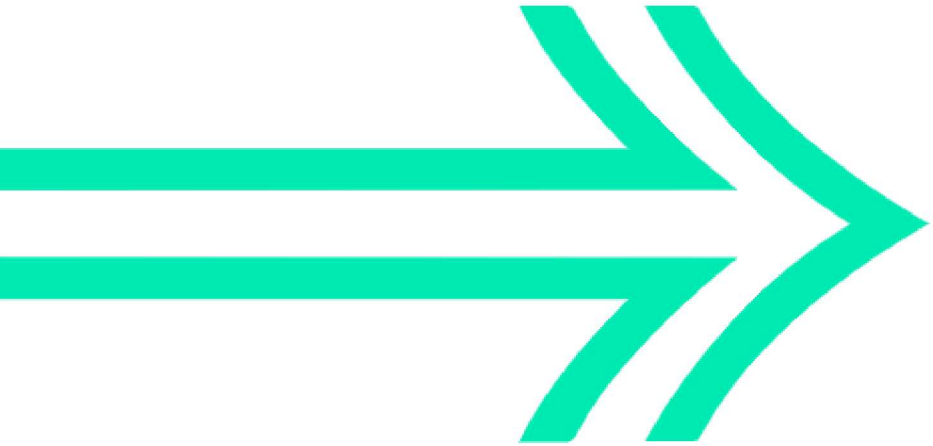
When the server is running, open a browser and navigate to:



Using JSON Server

We can create some State and a form to allow the user to input the Data.

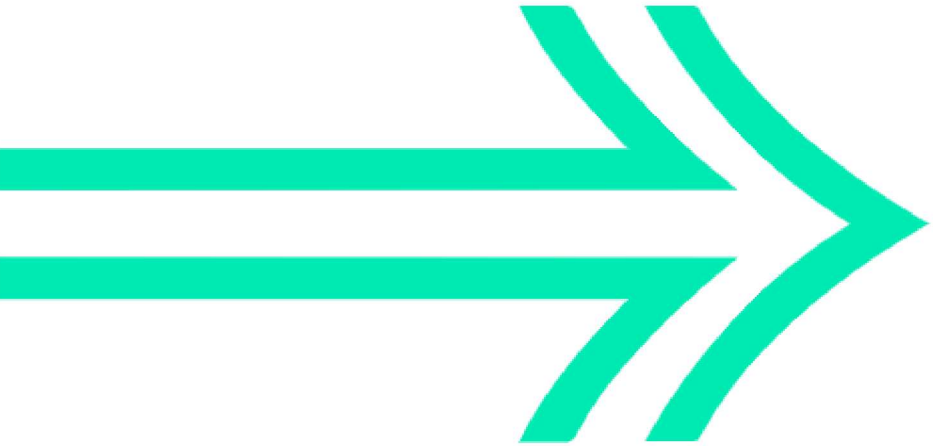
```
const Create = () => {

    const [first, setFirst] = useState('');
    const [second, setSecond] = useState('');
    const [title, setTitle] = useState('');
```



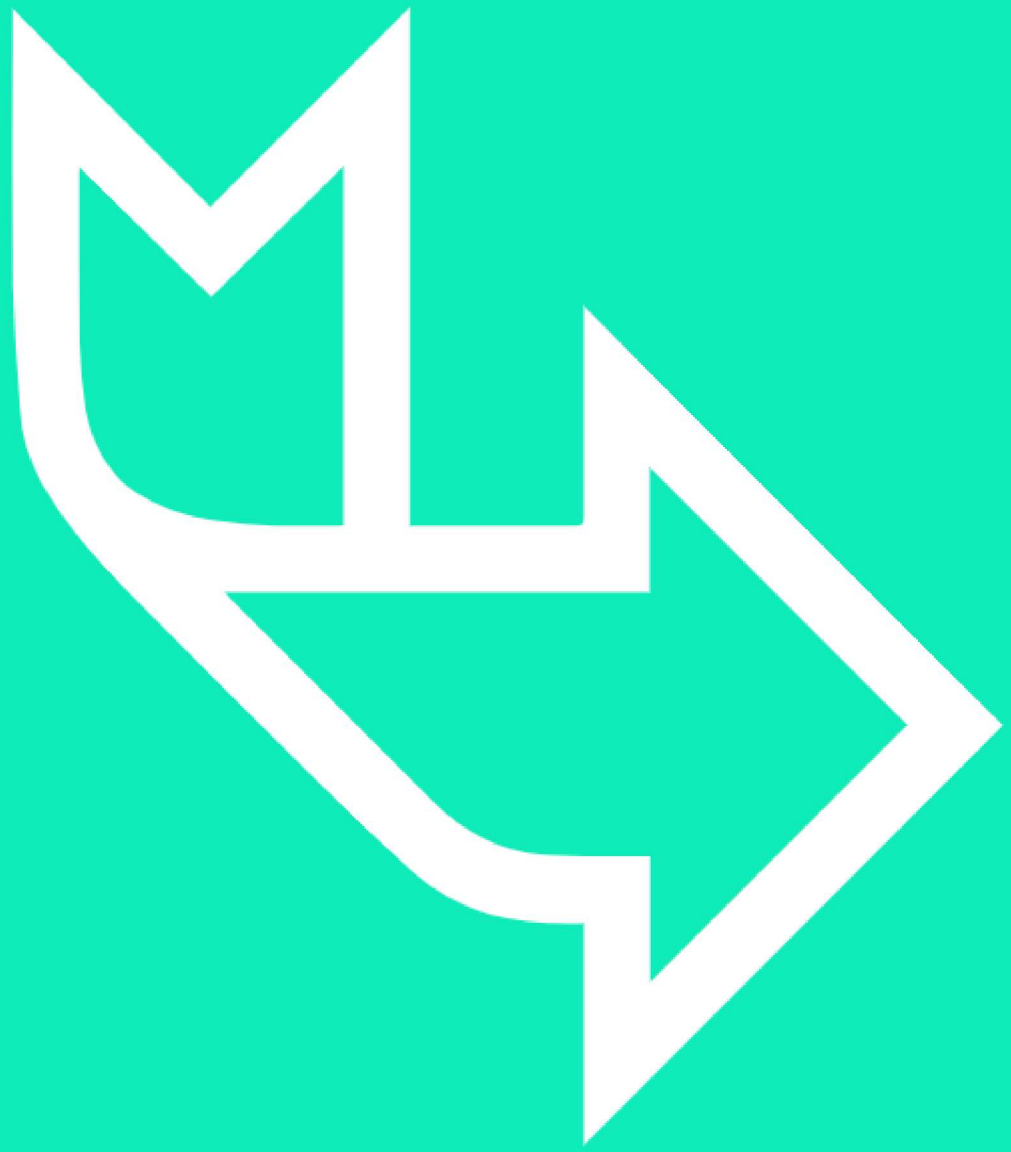
```
return (
    <form onSubmit={handleSubmit}>
        <label>First Name:</label>
        <input type="text"
        required
        value={first}
        onChange={(e) => setFirst(e.target.value)}
        />
        <br />
        <label>Last Name:</label>
        <input type="text"
        required
        value={second}
        onChange={(e) => setSecond(e.target.value)}
        />
        <br />
        <label>Title:</label>
        <select
        value={title}
        onChange={(e) => setTitle(e.target.value)}>
            <option value="mr">Mr</option>
            <option value="mrs">Mrs</option>
            <option value="dr">Dr</option>
        </select>
        <br />
        <button>Add User</button>
    </form>
);
```

Using JSON Server

Finally, we can use the JSON server to POST the form data to the json file. You can watch it update when you click the submit button.

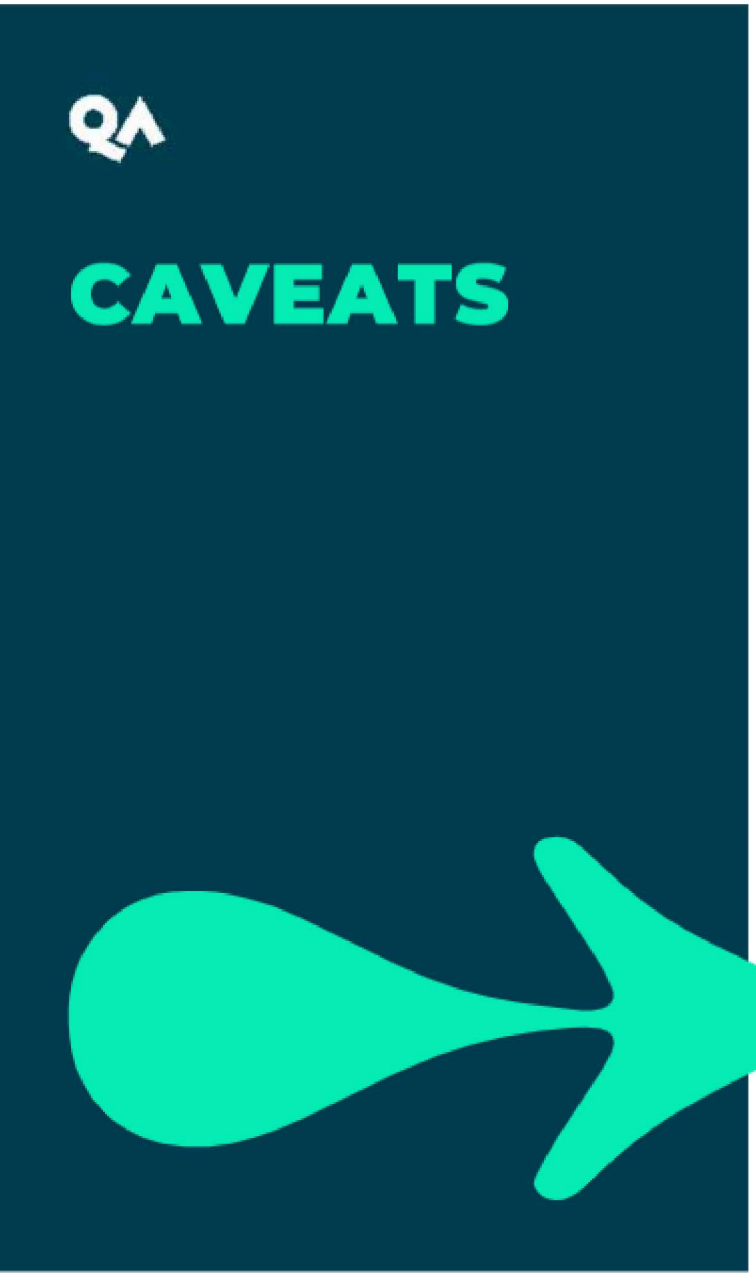


```
const handleSubmit= (e) => {
    e.preventDefault();
    const user = { first, second, title }

    fetch('http://localhost:8000/users', {
        method: 'POST',
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify(user)
    })

    setFirst('');
    setSecond('');
    setTitle('');
}
```



QuickLab 12

Editing Json data with react



We just saw **data fetching inside a component** with the useEffect hook.

Although our solution works, and is fine for very simple cases, in an application that is any more complex, this manual way of working has flaws.

Applications that use external data will generally need to

- cache data (tricky).
- track loading states.
- avoid duplicate requests.
- handle errors.

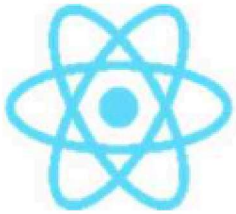
If you tried to implement all of this yourself, you'd be reinventing the wheel. These problems have already been solved by third-party libraries (e.g., react-query).

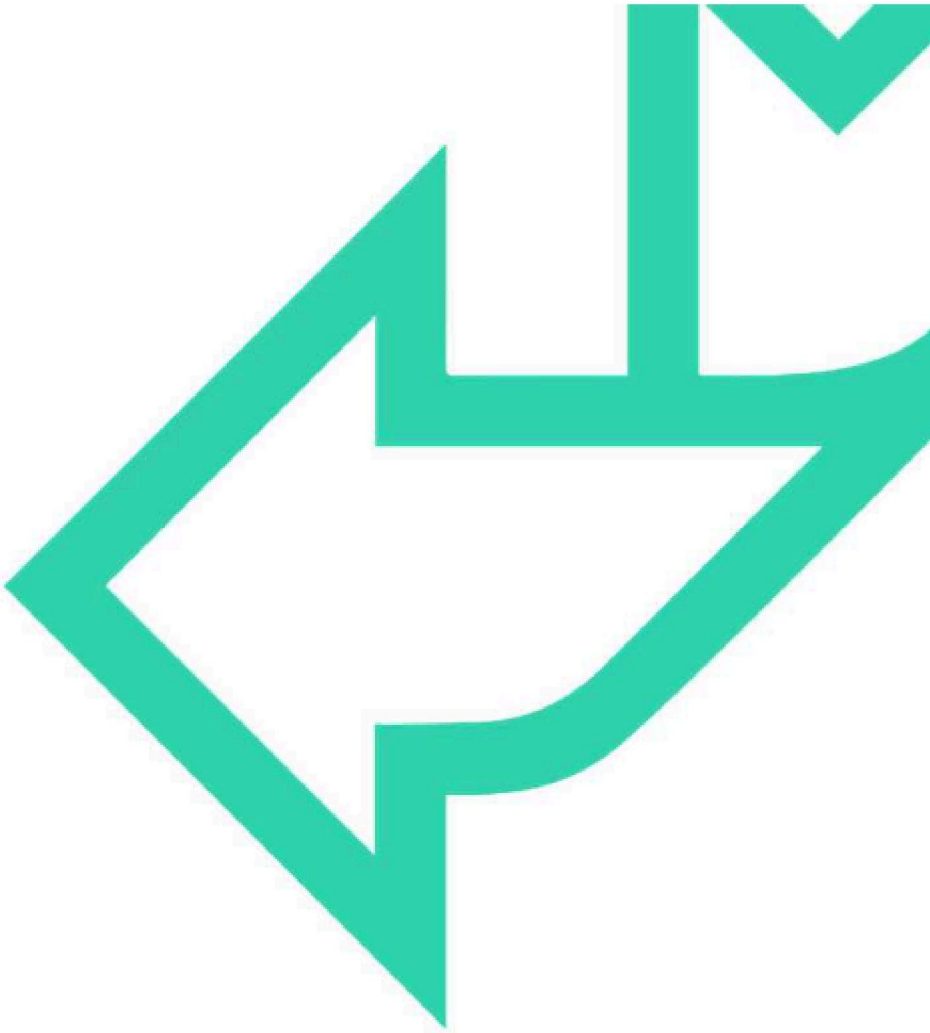
HANDS-ON PROJECT

At your own pace, work through:

- Challenge 1
- Challenge 2
- Challenge 3

QA POWERING POTENTIAL



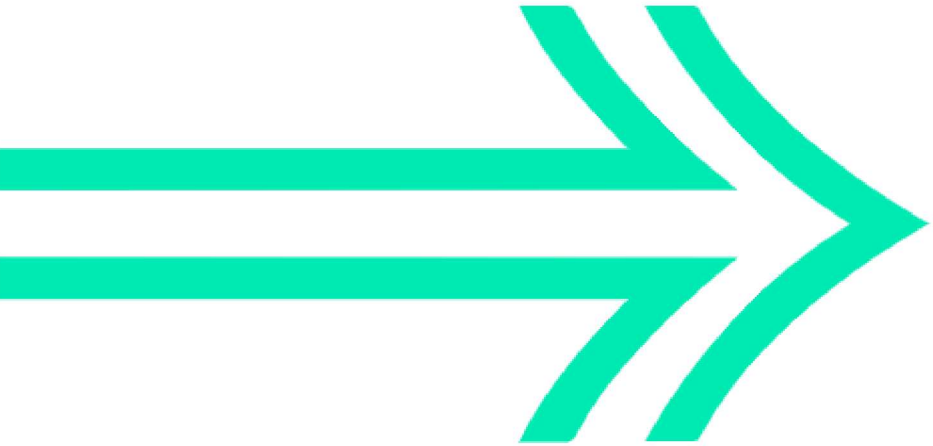


HANDS-ON WITH REACT

React Context

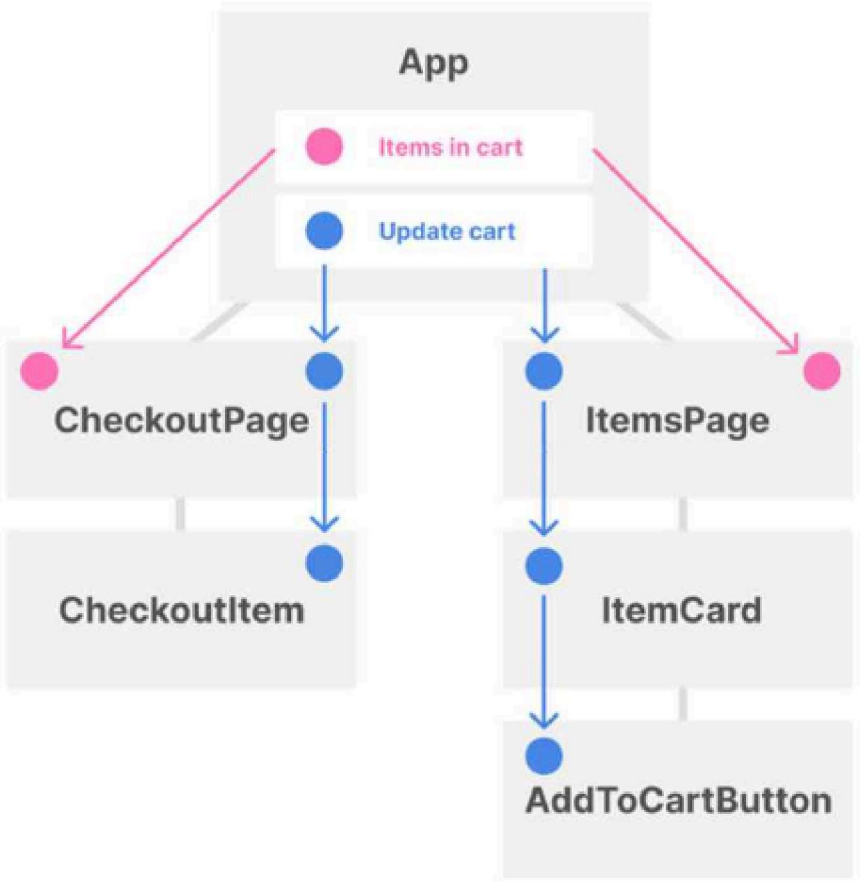
We can pass values to child components via props. However, things can become difficult to manage when you start passing props down multiple nested layers. This is known as "**prop drilling**".

A React feature called "**Context**" is one way to solve this. Context lets you pass data deep down layers of components.



Let's use an example to demonstrate this concept.

Without Context



In this example, we have a shop app that stores items in the user's shopping cart.

Deep components such as `AddToCartButton` need to be able to update the cart, which means the functionality needs to be **manually passed through multiple layers of components** via props.

Also, the functionality is passed through components that don't care about it, such as `ItemsPage`.

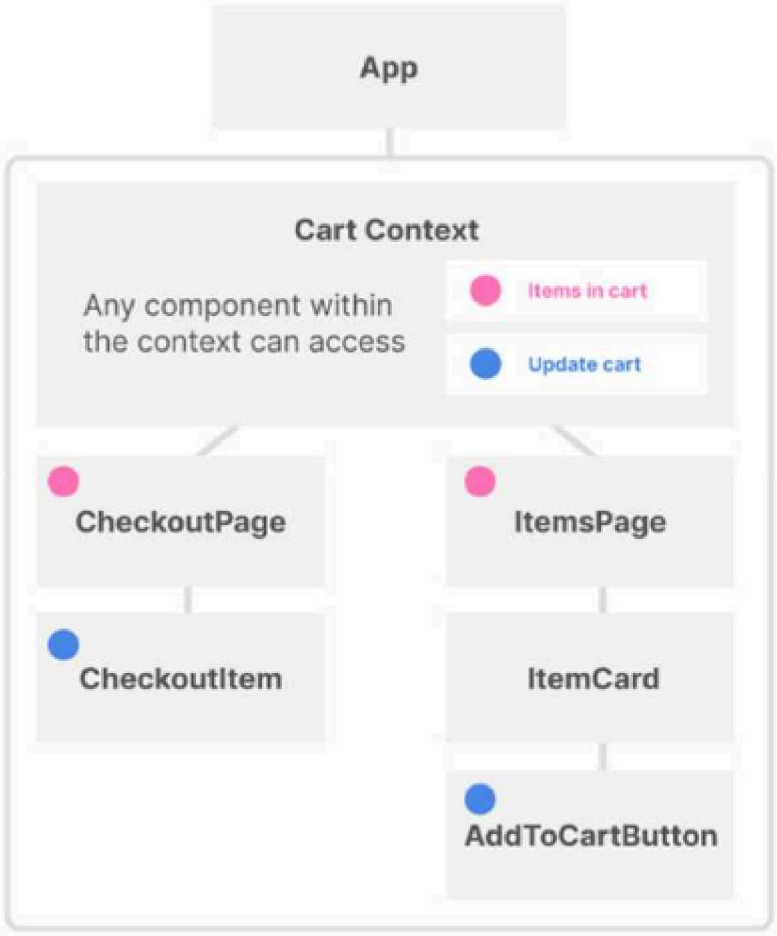
Pros

- Simple to implement

Cons

- Clunky if your app has deeply nested components that require data from above

With Context



With Context, we no **longer need to pass data through many layers of components**. Context lets us send values to components far down the tree.

A context can hold whatever values or functions you want. **Any component within the context has access** to these values.

`AddToCartButton` now has easy access to the update cart functionality because it is inside the cart context.

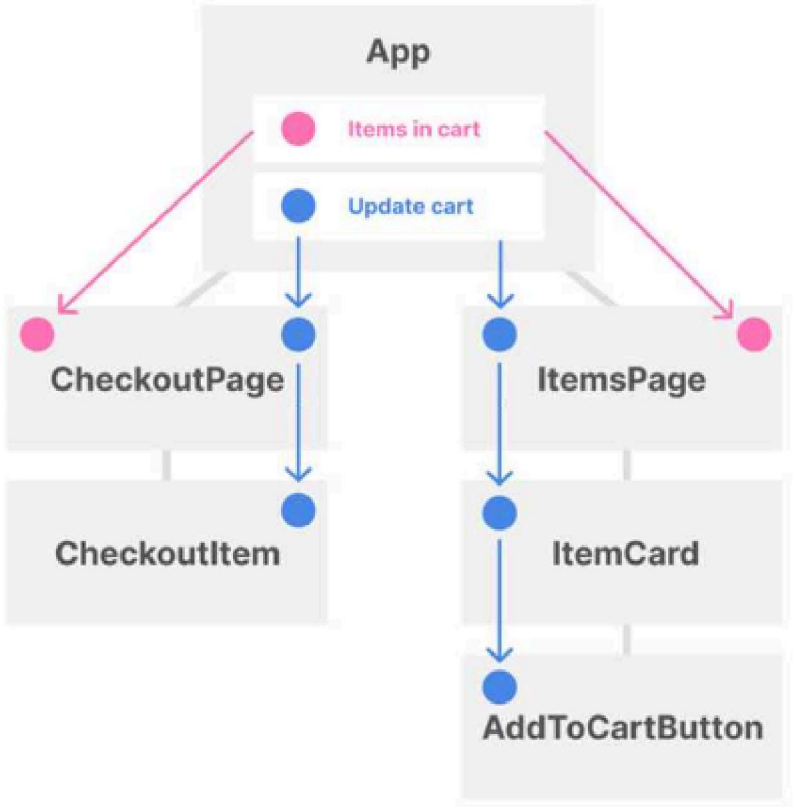
Pros

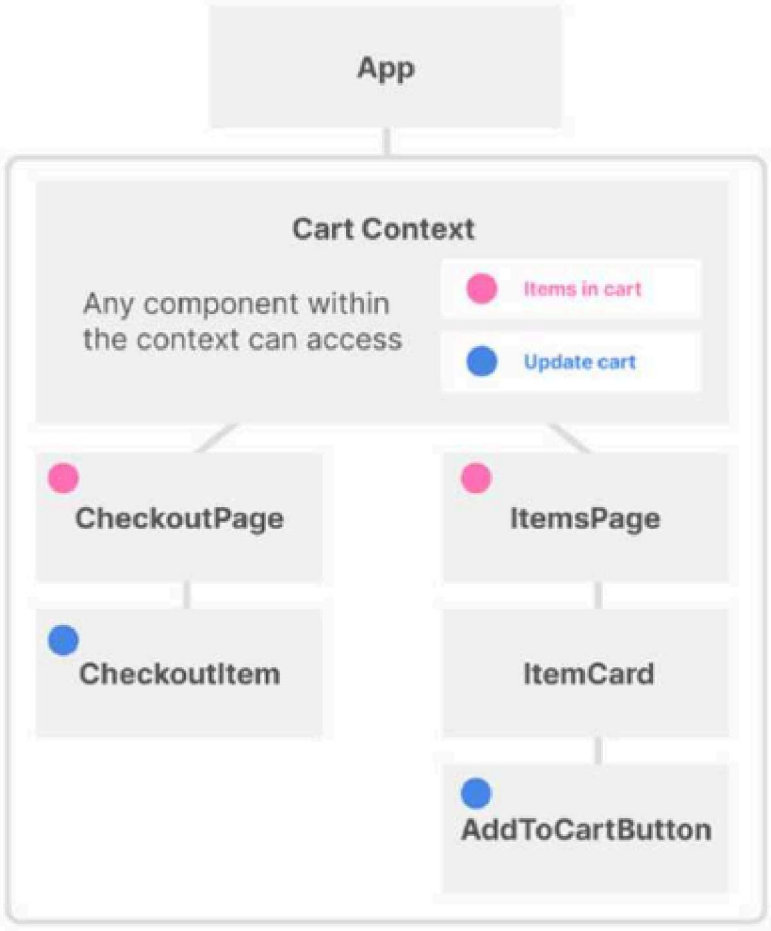
- Eliminates need for prop drilling
- More maintainable

Cons

- Takes a few more steps to get set up
- Can cause unwanted re-renders

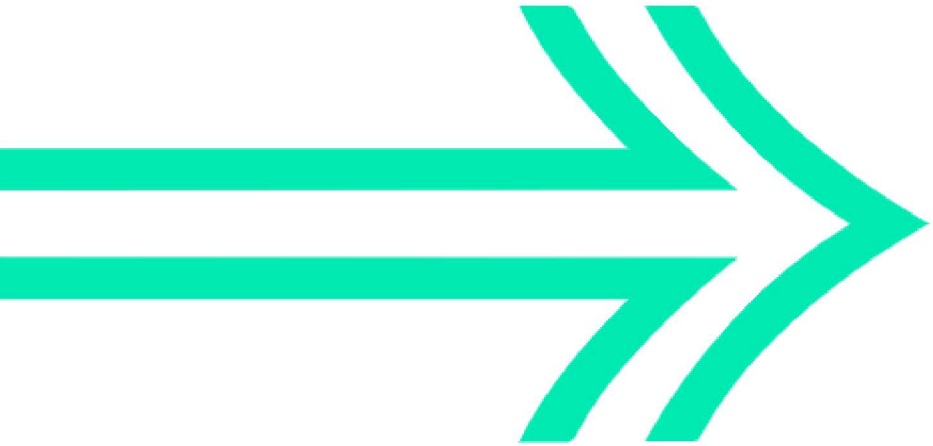
Before and after





React Context

Now that we've looked at why Context is useful and what it can do, let's see this in practice.



React Context

We need to make sure we have imported createContext from React:

```
import React, { createContext, useState } from 'react'
import itemsData from '../itemsData.json'
```

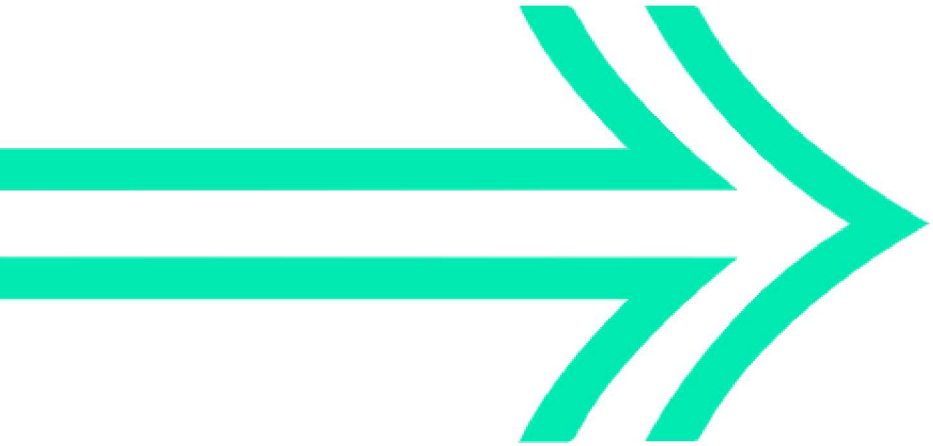
We create a context that can be passed to the children.

```
export const CartContext = createContext()

export function CartProvider({ children }) {
  const [itemsInCart, setItemsInCart] = useState([])

  function addToCart(itemId) {
    const item = itemsData.find((item) => item.id === itemId)
    setItemsInCart((prev) => [...prev, item])
  }

  const contextValue = {
    itemsInCart,
    addToCart
  }

  return (
    <CartContext.Provider value={contextValue}>{children}</CartContext.Provider>
  )
}
```

React Context

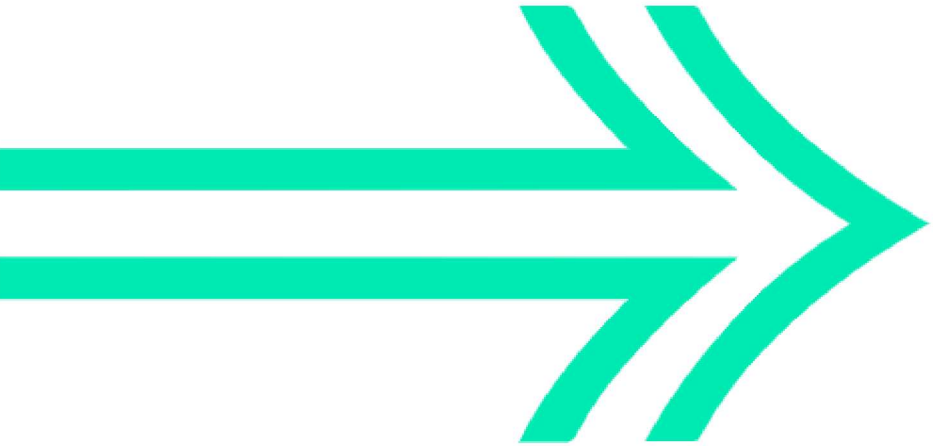
In the Item Cart, we import the context created and ensure we have imported the useContext hook.

```
import React, { useContext } from 'react'
import { CartContext } from '../context/cart-context'

export default function ItemCard({ id, symbol, name, price }) {
  const { addToCart } = useContext(CartContext)

  function handleAddToCart() {
    addToCart(id)
  }

  return (
    <div className="item-card">
      <div className="symbol">{symbol}</div>
      <h3>{name}</h3>
      <p>£{price.toFixed(2)}</p>
      <button onClick={handleAddToCart}>Add to cart</button>
    </div>
  )
}
```



When the call to action button is clicked, the addToCart context is accessed and run like a normal function call.

React Context

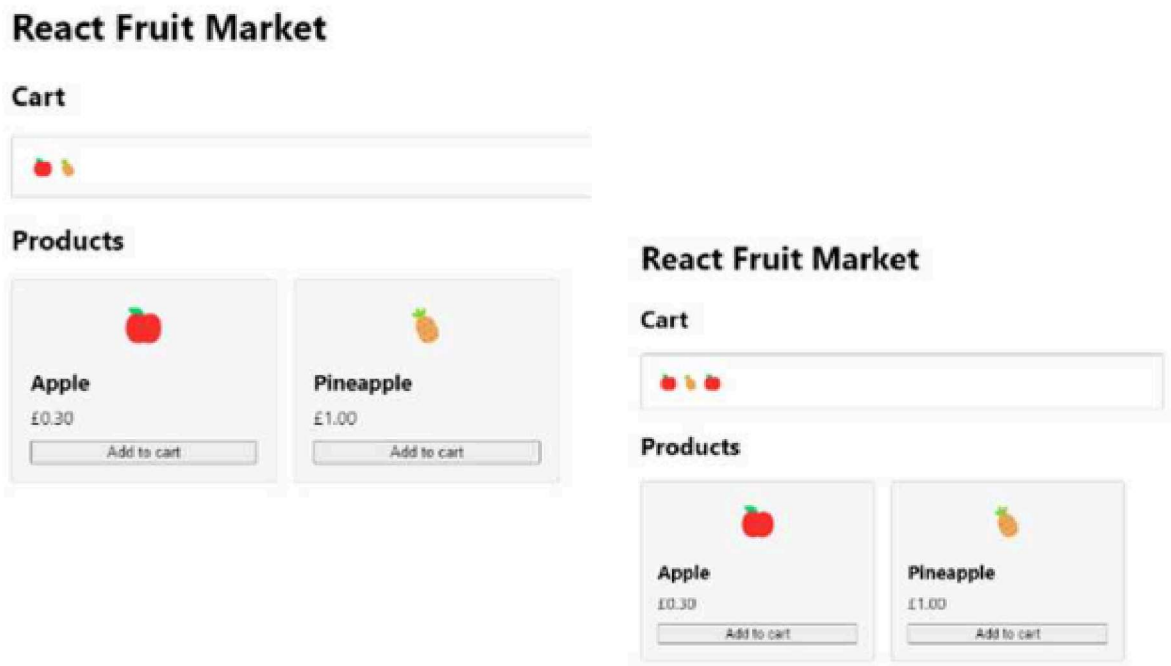


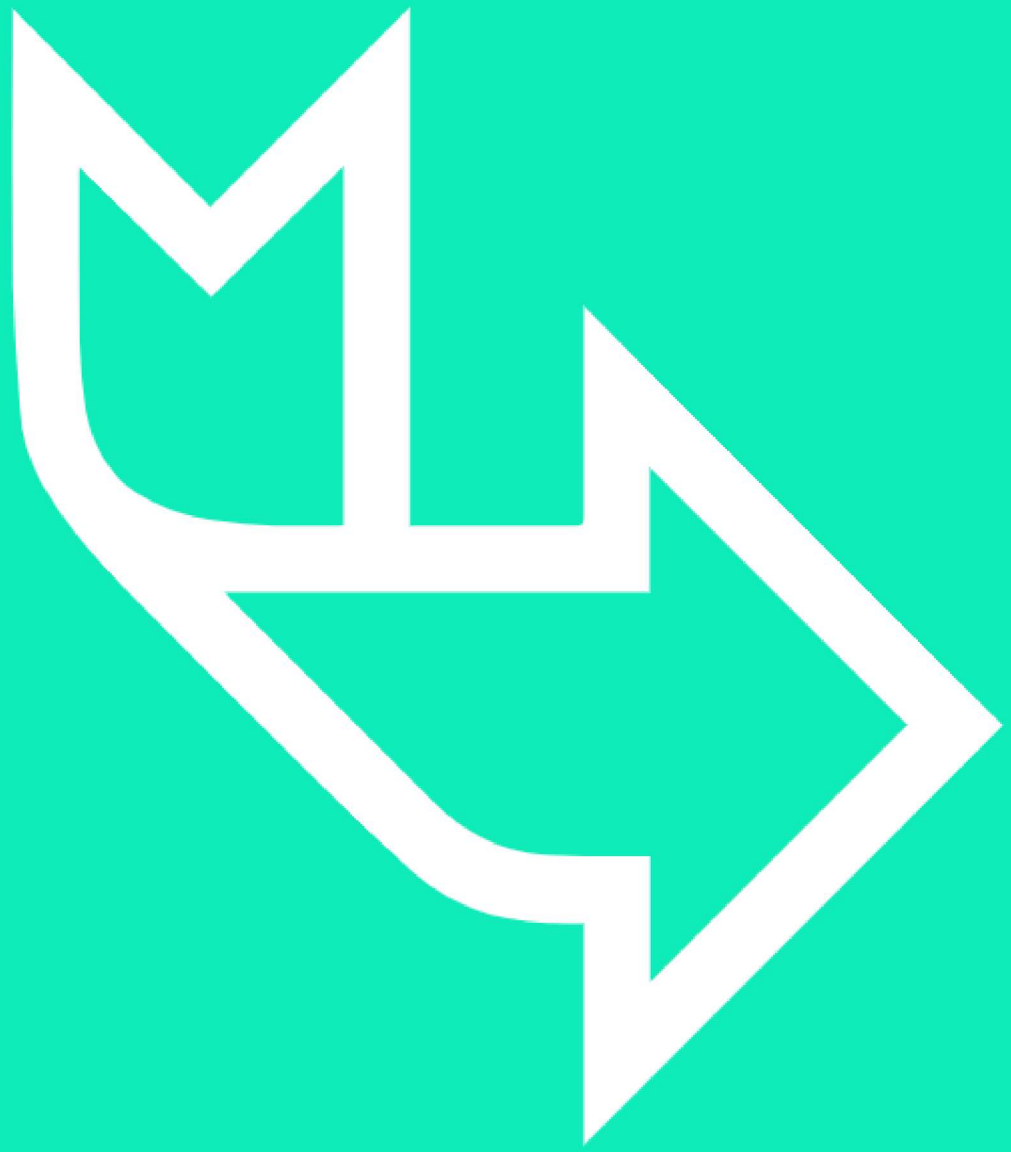
This updates the Cartlist items which re-renders in the DOM:

```
import React, { useContext } from 'react'
import { CartContext } from '../context/cart-context'

export default function CartList() {
  const { itemsInCart } = useContext(CartContext)

  return (
    <div>
      <h2>Cart</h2>
      <div className="cart-wrapper">
        {itemsInCart.map((item) => (
          <span>{item.symbol}</span>
        ))}
      </div>
    </div>
  )
}
```

React Context

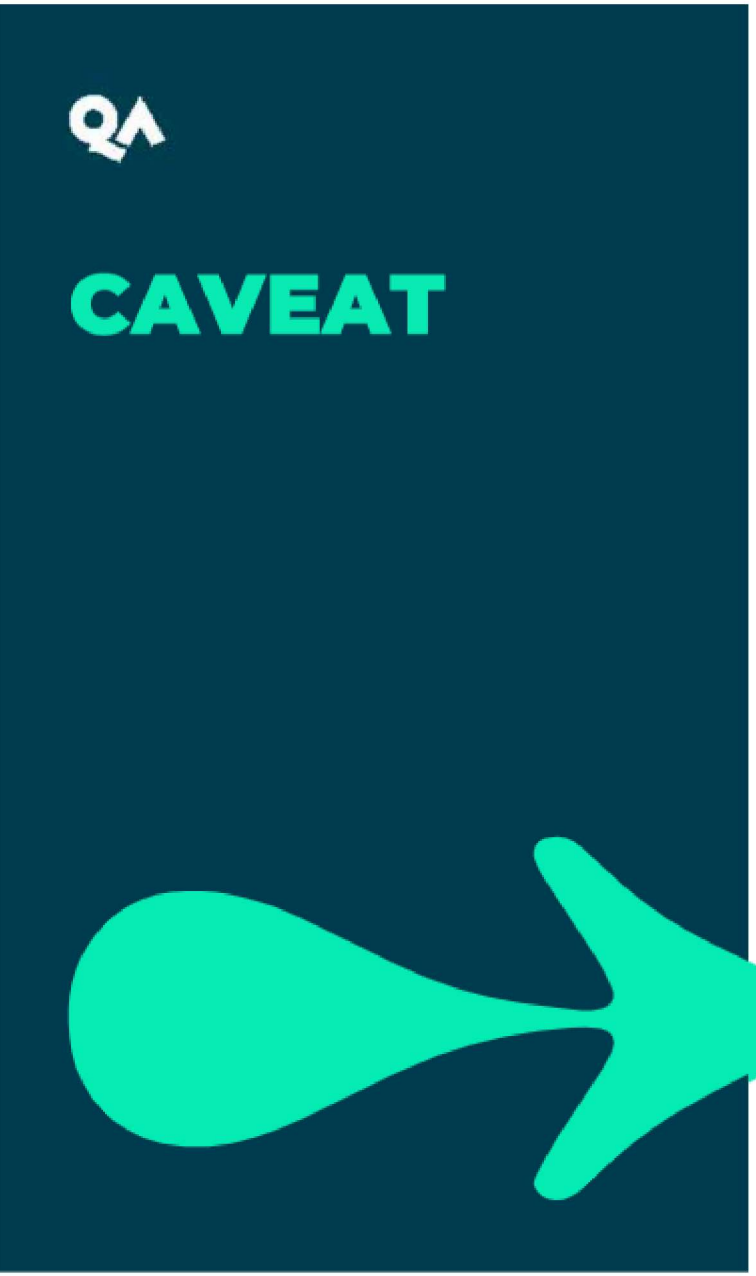
Updating the cart after every click using the context.





Quick Lab 13

Context



We just saw how Context can aid in managing state in your application.

Many **state management libraries** also exist to solve the problem of managing application state.

Many real-world applications utilise these libraries, because they solve a lot of problems, such as:

- providing centralised, global state.
- tracing state (useful for debugging).
- decoupling state logic from the UI (useful for testing).

Some examples of state management libraries are *Redux Toolkit* and *Mobx*.

We don't cover these in this module, but it's worth being aware that they exist because they are so commonly used.

HANDS-ON PROJECT

At your own pace, work through:

- Challenge 1
- Challenge 2
- Challenge 3
- Challenge 4
- Extensions

THAT'S A WRAP!