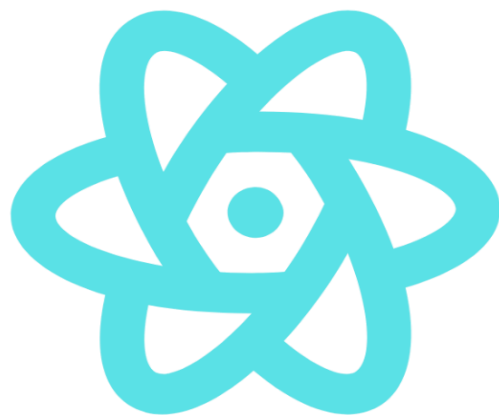


React Simplified



The Essential Guide for Beginners

Esther Vaati

Table of Contents

Introduction to React	3
What is React?	3
💡 Why React	3
💡 Getting Started with React	3
Step1: Set up Your HTML File	4
What is CreateElement	5
Create and Render an Element	5
Adding Properties.	6
Nesting Elements	6
What is JSX	9
Setting up a React environment	12
Recommended Tools for React Development:	12
Start a New React Project	14
Application Structure	14
JSX Expressions	19
What is <code>jsx</code> Fragment	21
Components	22
Create Your First Component	23
Styling in React	24
Props	26
Destructuring Props	32
List Items	32
Iterating with the <code>Map()</code> Method	33
Conditional Rendering	37
Other examples of Conditional Rendering	40
JSX Tags must be Closed	42
JSX Attributes must use CamelCase	42
Conditionals in JSX	44
State Management in React	44
Use the State in JSX	46
State management in Forms	46
How State Works in Forms	48
Lifting State in React	51
Managing Side Effects with the <code>useEffect</code> Hook	55
API Requests	55
The dependency array	56
Local Storage	57
Conclusion	60

Introduction to React

What is React?

React is an open-source JavaScript library developed by Facebook in 2013 for building user interfaces, particularly for web applications. Facebook developed React because they had a problem with the **state** of messages where read messages would still display the unread count.

React comes with its own rules, and to understand React, you need to adhere to these rules and understand how React renders when an application loads. Once you understand these concepts, learning React becomes a breeze

Why React

React is one of the most popular frameworks and one of the most in-demand libraries. It also relies heavily on JavaScript, so if you are proficient in JavaScript, learning React will prove to be easier.

React is also an evolving library with a strong vibrant, active community who are continuously improving its features.

React operates on the principle of reusable components. Unlike in a traditional setting where you would have an **index.html** file with all the code, React allows you to have components for each section, for example, you can have a **Header** component, a **Sidebar** component, a **Main** component, and a Footer component and so on.

Getting Started with React

While it's common to build in React by using a framework, you can use a **CDN** (Content Delivery Network) to start building in with React. This approach allows you to understand how React works under the hood.

Step1: Set up Your HTML File

Create an HTML file with the basic structure, including `<!DOCTYPE html>`, `<html>` `</html>`, `<head>` `</head>`, and `<body>` `</body>` and the necessary scripts and the CDN links.

Load both **React** and **ReactDOM** with CDN links. By using a CDN, you can start building with React without installing any extra tools on your local development environment.

In this example, we will use the unpkg to load React.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <title>React CDN Example</title>
  <!-- React Library -->
  <script
src="https://unpkg.com/react@17/umd/react.development.js"></script>
  <!-- React DOM Library -->
  <script
src="https://unpkg.com/react-dom@17/umd/react-dom.development.js"></s
cript>
</head>
<body>
  <div id="root"></div>
  <script>
    // Your React code will go here
  </script>
</body>
</html>
```

In the head section, we have two script tags for loading **React** and **ReactDOM** from a CDN. Then a div with the **id = root**. This is where our react application will be rendered.

What is CreateElement

CreateElement is a function used by React to create React elements. React elements are the building blocks of **React**. The basic syntax of creating a React element with the **createElement** function looks like this:

```
React.createElement(type, [props], [...children])
```

- **type** - this can be a string for HTML elements (i.e. "div", "h1"), etc.
- **props** : this is an object containing the properties to be passed to the element. For example {id:" container"} might be passed to a div element.
- **children** : this is the content inside the element.

Create and Render an Element

Let's start by creating a simple H1 heading with the content " Hello React ". In the script tag, add the code below.

```
const element = React.createElement('h1', {}, "Hello World!");  
ReactDOM.render(element, document.getElementById('root'));
```

When you open the **index.html** file with your browser, you can see the content rendered on your browser.

Hello World!

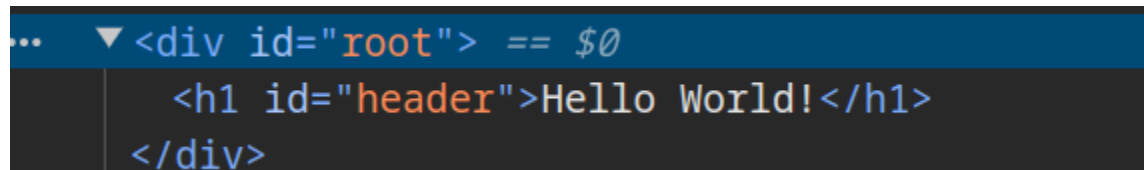
Congratulations, you have just created your first React App.

Adding Properties.

Our H1 element at the moment does not have any properties, let's fix that, add an **id == header**

```
const element = React.createElement('h1', {id : "header"}, "Hello World!");
ReactDOM.render(element, document.getElementById('root'));
```

Now, if you inspect with Developer tools, you can see the ID has been applied.

A screenshot of a web browser's developer tools component inspector. It shows a tree view of the rendered DOM. The root element is a <div id="root"> with a value of \$0. Inside this div is an <h1 id="header">Hello World!</h1> element. The </div> tag is also visible below the h1 element.

```
... <div id="root"> == $0
  <h1 id="header">Hello World!</h1>
</div>
```

Nesting Elements

You can also nest **createElement** call to create more elements. For example, let's add more elements and enclose them with a div.

```
const element = React.createElement('h1', {id : "header"}, "Hello World!");
const element2 = React.createElement('p', {id : "paragraph"}, "This is a paragraph.");

const divElement = React.createElement('div', {id : "container"}, element, element2);

ReactDOM.render(divElement, document.getElementById('root'));
```

`const` `divElement` = `React.createElement('div', {id: "container"}, element, element2);` This creates a `<div>` element with the ID "container". Inside this div, we are passing the 2 elements as **children**.

```
▼ <div id="root">
...  ▼ <div id="container"> == $0
      <h1 id="header">Hello World!</h1>
      <p id="paragraph">This is a paragraph.
      </p>
    </div>
  </div>
```

Here is the final code

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <title>React CDN Example</title>
  <!-- React Library -->
  <script
src="https://unpkg.com/react@17/umd/react.development.js"></script>
  <!-- React DOM Library -->
  <script
src="https://unpkg.com/react-dom@17/umd/react-dom.development.js"></s
cript>
</head>
<body>
  <div id="root"></div>
  <script>
```

```

    const element = React.createElement('h1', {id
:"header"}, "Hello World!");
    const element2 = React.createElement('p', {id : "paragraph"},
"This is a paragraph.");
    const divElement = React.createElement('div', {id
:"container"}, element, element2);
    ReactDOM.render(divElement, document.getElementById('root'));
  </script>
</body>
</html>

```

This method of creating a React application is a great way to understand how React works under the hood but for larger applications, you would typically use **JSX**. JSX syntax is the recommended way to create React applications

JSX is a syntax extension for JavaScript that lets you write HTML-like markup inside JavaScript files. JSX is easier to read and write because it utilizes pure JavaScript. So if you have a solid foundation in JavaScript, Learning React will not be that difficult.

For example, if we were to use a component, the previous code we have just written in JSX, we will have something like this;

```

function MyComponent() {
  return (
    <div id="container">
      <h1 id="header">Hello World!</h1>
      <p id="paragraph">This is a paragraph.</p>
    </div>
  );
}

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

```

As you can see, this code is more concise

What is JSX

Normally, when creating web applications, you write all your HTML in one file and the JavaScript logic in a .js file. However, this is not the case with **JSX**. With JSX, all the HTML and Javascript live in one file

So let's convert our code to use JSX

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width,
initial-scale=1.0" />
    <title>React CDN Example</title>
    <!-- React Library -->
    <script
src="https://unpkg.com/react@17/umd/react.development.js"></script>
    <!-- React DOM Library -->
    <script
src="https://unpkg.com/react-dom@17/umd/react-dom.development.js"></s
cript>
  </head>
  <body>
    <div id="root"></div>
    <script >
      const name = "World";
      const message = "This is a dynamic paragraph.";

      const element = (
        <div id="container">
          <h1 id="header">Hello !</h1>
          <p id="paragraph">Hello</p>
        </div>
      );

      ReactDOM.render(element, document.getElementById("root"));
    </script>
  </body>
```

```
</html>
```

This code will not work because the JSX code needs to be compiled. Compiling is important because browsers don't understand JSX code, so tools like **Babel** transform the code into a format suitable for browsers to read.

To fix this issue, include a compiler such as Babel and specify that the JSX code should be transpiled. Update the code as follows:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width,
initial-scale=1.0" />
    <title>React CDN Example</title>
    <!-- React Library -->
    <script
src="https://unpkg.com/react@17/umd/react.development.js"></script>
    <!-- React DOM Library -->
    <script
src="https://unpkg.com/react-dom@17/umd/react-dom.development.js"></s
cript>
    <!-- Babel for JSX -->
    <script
src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
  </head>
  <body>
    <div id="root"></div>
    <script type="text/babel">
      const name = "World";
      const message = "This is a dynamic paragraph.";

      const element = (
        <div id="container">
          <h1 id="header">Hello {name}!</h1>
          <p id="paragraph">{message}</p>

```

```
    </div>

    );

    ReactDOM.render(element, document.getElementById("root"));
  </script>
</body>
</html>
```

The line `<script`

`src="https://unpkg.com/@babel/standalone/babel.min.js"></script>` loads Babel, which allows JSX to be compiled in the browser.

Change the script tag containing JSX to `type="text/babel"`. This ensures that Babel knows how to process it.

Now our code is working as expected.

You might have noticed when we injected the javascript variable inside our **HTML** markup, we used curly braces.

```
<h1 id="header">Hello {name}!</h1>
<p id="paragraph">{message}</p>
```

In **JSX**, if you need to add JavaScript expressions or variables inside your HTML-like markup, you need to wrap the JS code inside curly braces `{ }`. The curly braces tell JSX that the code inside the braces is Javascript code and not plain text.

Setting up a React environment

We have seen how to create a **React** app using a CDN, along with **createElement**, **ReactDOM**, and Babel. However, this method is not recommended for production applications, as it does not scale well for larger projects.

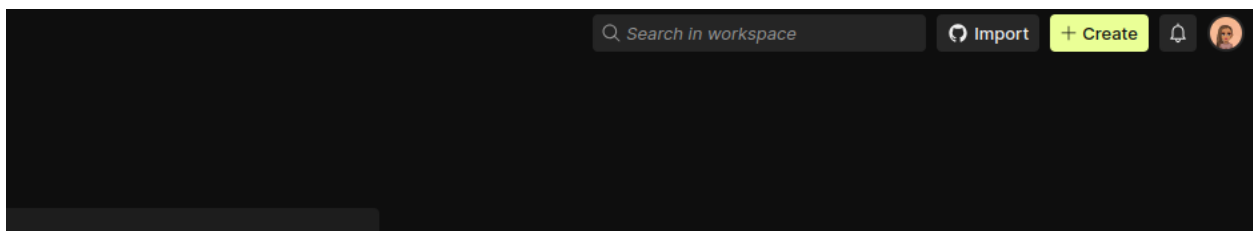
Luckily, several frameworks and tools exist that provide built-in support for **React** and abstract away the complexities of setting up a build process. These tools enable developers to focus on writing code rather than managing configurations.

Recommended Tools for React Development:

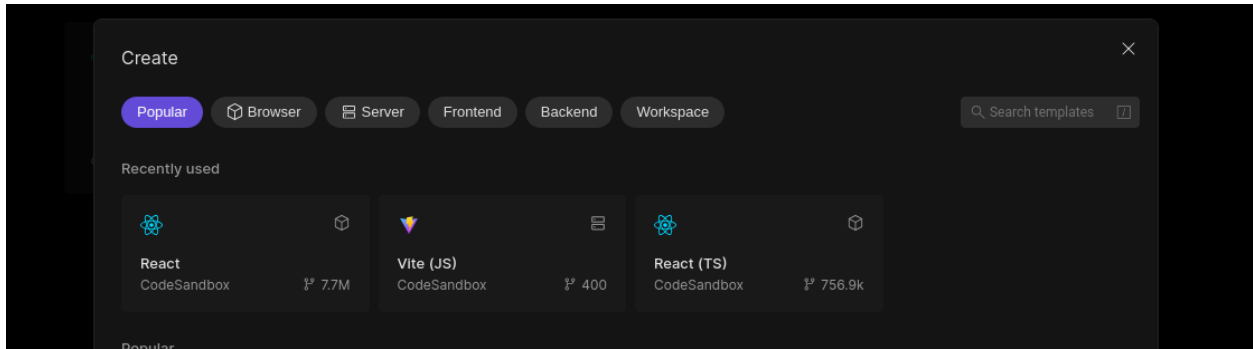
1. **Vite**: Vite is a modern build tool that provides a fast and efficient development environment for React applications.
2. **Create React App (CRA)**: Create React App is a widely used tool that sets up a new React project with a solid default configuration.

In this guide, we will use Create React App to create our React applications. If you don't fancy a local development environment, you can use <https://codesandbox.io> to host your code.

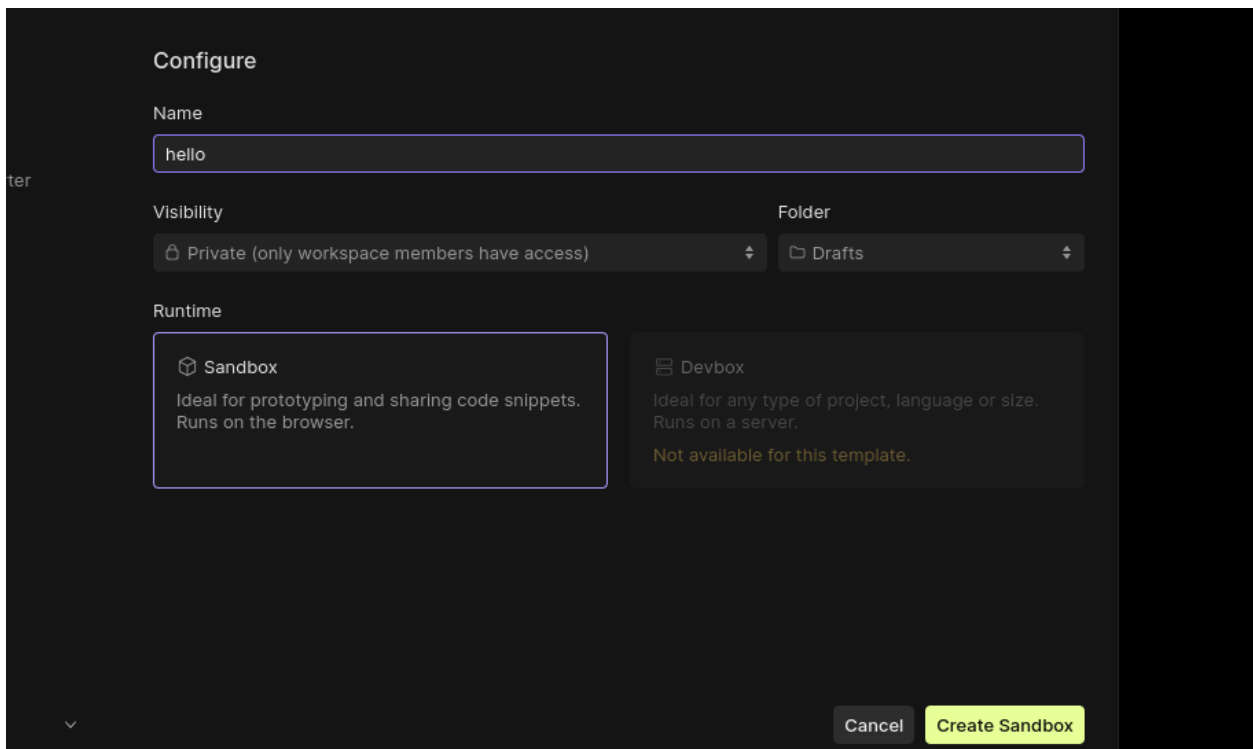
Head over to <https://codesandbox.io/> and create an account. Once you are logged in. On your dashboard, create a new React application using Vite.



In the next screen, select React



Provide a name for your project and click the Create Sandbox button.



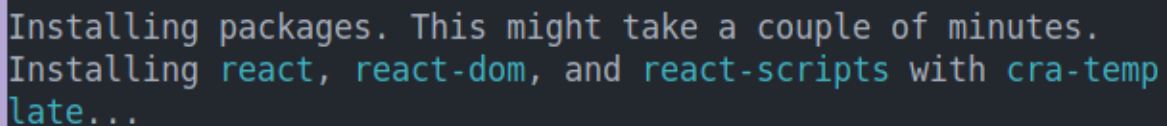
If you prefer a local environment, then you should have Node.js installed. React requires Node.js to run its development server and build the project. You can download the latest stable version of Node.js from [Node.js official website](https://nodejs.org/en/).

Start a New React Project with Create-react app

To create a new project, issue the npx create-react app command.

```
npx create-react-app hello-react
```

You should see something like this:



```
Installing packages. This might take a couple of minutes.  
Installing react, react-dom, and react-scripts with cra-temp  
late...
```



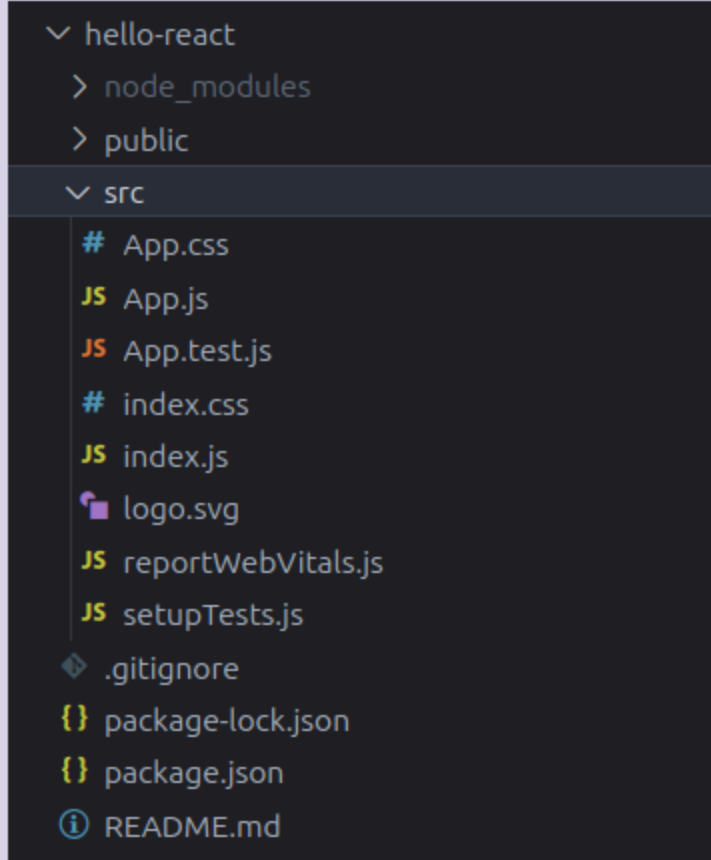
Once the installation is complete, cd into the app and issue the npm start command

```
cd hello-react  
npm start
```

Your app should now be running at <http://localhost:3000/>

Application Structure

The app structure looks like this:



Whether you use a local development environment or an online tool like [CodeSandbox](#), the structure of the application will be the same. Both methods provide a similar setup, with files like **index.js**, **App.js**, and **index.html** acting as the foundation of your React app.

In this guide, we will use CodeSandbox but for larger applications, a local development environment offers more control and is preferred.

In the public folder, we have an **index.html** file which looks like this:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1, shrink-to-fit=no">
  <meta name="theme-color" content="#000000">
  <link rel="manifest" href="%PUBLIC_URL%/manifest.json">
  <link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico">
  <title>React App</title>
</head>
<body>
  <noscript>
    You need to enable JavaScript to run this app.
  </noscript>
  <div id="root"></div>

</body>
</html>
```

The **index.html** file is a standard HTML file, except that it includes a **<div id="root"></div>**. In React, this **<div id="root"></div>** serves as the mounting point where React renders your application.

All the content will be injected into this div. Unlike in a traditional app, where you would select each element using methods like **getElementById** or **querySelector**, React uses the **ReactDOM.render()** method to render components into this div. This process is handled in the **index.js** file.

In your **index.js** file, you have the code below

```
import { StrictMode } from "react";
import { createRoot } from "react-dom/client";
import App from "./App";
const rootElement = document.getElementById("root");
const root = createRoot(rootElement);

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

Here, we have imported **StrictMode** from **react** and **createRoot** from **react-dom/client**. **StrictMode** is a feature in React used to catch potential bugs and errors in your application. It helps enforce best practices like warning about deprecated features, checking for side effects, and verifying component lifecycle methods.

const rootElement = document.getElementById("root"); is a reference to the div with the **id == root** from the **index.html**. Unlike JavaScript where we use a DOM, React uses a virtual DOM to render components.

The virtual DOM allows React to make updates more efficiently by calculating changes in memory and then updating only the necessary parts of the actual DOM.

Once we get a reference to the root, we call **createRoot** to create a React root. The React root will display content in the browser. Lastly, we render the App component.

This is the App component in the app.js file.

```
import "./styles.css";

export default function App() {
  return (
    <div className="App">
      <h1>Hello CodeSandbox</h1>
      <h2>Start editing to see some magic happen!</h2>
    </div>
  );
}
```

The App component introduces us to the **JSX (JavaScript XML)** syntax. React uses JSX syntax to describe the UI. Most of the code in React applications is written in JSX. As you can see, JSX looks similar to regular HTML. Behind the scenes, JSX is transformed into JavaScript function calls (e.g., **React.createElement**) before being rendered to the DOM.

At the top of the file, we can see `import "./styles.css";` statement that imports a CSS file that styles the component. In a regular HTML file, we use class to apply style, but in JSX, the equivalent is **className**. The `className="App"` is used to apply CSS styles defined in the styles.css file. In JSX, class is a reserved keyword in JavaScript, so it can't be used as an attribute name.

To prevent conflicts with the JavaScript language, JSX uses **className** instead of class to define CSS classes on elements.

A **React** component is a regular javascript function or class that returns JSX. In the return statement, we have the JSX code that describes the UI. The UI has a div element with two headings: an `<h1>` and an `<h2>`.

The `export default` keyword is important because it allows us to export a component (like App) as the default export of a file. This means that when another file imports **App**, it can do so without using curly braces `{}` around the component name.

In our case, we have seen the App component imported in the **index.js** file.

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

import App from "./App";

const rootElement = document.getElementById("root");
const root = createRoot(rootElement);

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

If you don't use export default, then you would then have to use a named export like this

```
import { App } from './App';
```

JSX Expressions

Earlier, we saw that we need to enclose any Javascript code with curly braces { }

Let's do more practice.

Create the following variables in the **App.js** file.

```
greeting ("Hello codeSandbox")
message (e.g., "Start editing to see some magic happen
.")
```

Inject them into **h1** and **p** tags respectively. The elements should be contained in a div element. The App component now looks like this.

```
import "./styles.css";

export default function App() {
  const greeting = "Hello CodeSandbox";
  const message = "Start editing to see some magic happen! ";
  return (
    <div className="App">
      <h1>{greeting}</h1>
      <h2>{message}</h2>
    </div>
  );
}
```

Let's duplicate the div with the **className = App**

```
import "./styles.css";

export default function App() {
  const greeting = "Hello CodeSandbox";
  const message = "Start editing to see some magic happen! ";
  return (
    <div className="App">
      <h1>{greeting}</h1>
      <h2>{message}</h2>
    </div>
    <div className="App">
      <h1>{greeting}</h1>
      <h2>{message}</h2>
    </div>
  );
}
```

When we do this, we run into some errors, The error message "Adjacent JSX elements must be wrapped in an enclosing tag." pops up

TypeError

Cannot assign to read only property 'message' of object
'SyntaxError: /src/App.js: Adjacent JSX elements must be wrapped in an enclosing tag. Did you want a JSX fragment <>...</>? (11:5)

This error occurs because, in **React**, elements must be wrapped in a single parent element. To fix this error, we can use a `div` as a wrapper for all the elements or use a JSX fragment.

What is jsx Fragment

A JSX **fragment** is a tool in React that lets you wrap multiple elements without adding any additional `div` elements. Instead of creating a wrapper `div` for all the `divs`, let's create a JSX fragment.

The syntax for a fragment looks like this:

shorthand syntax: `<>...</>`

full syntax: `<React.Fragment>...</React.Fragment>`

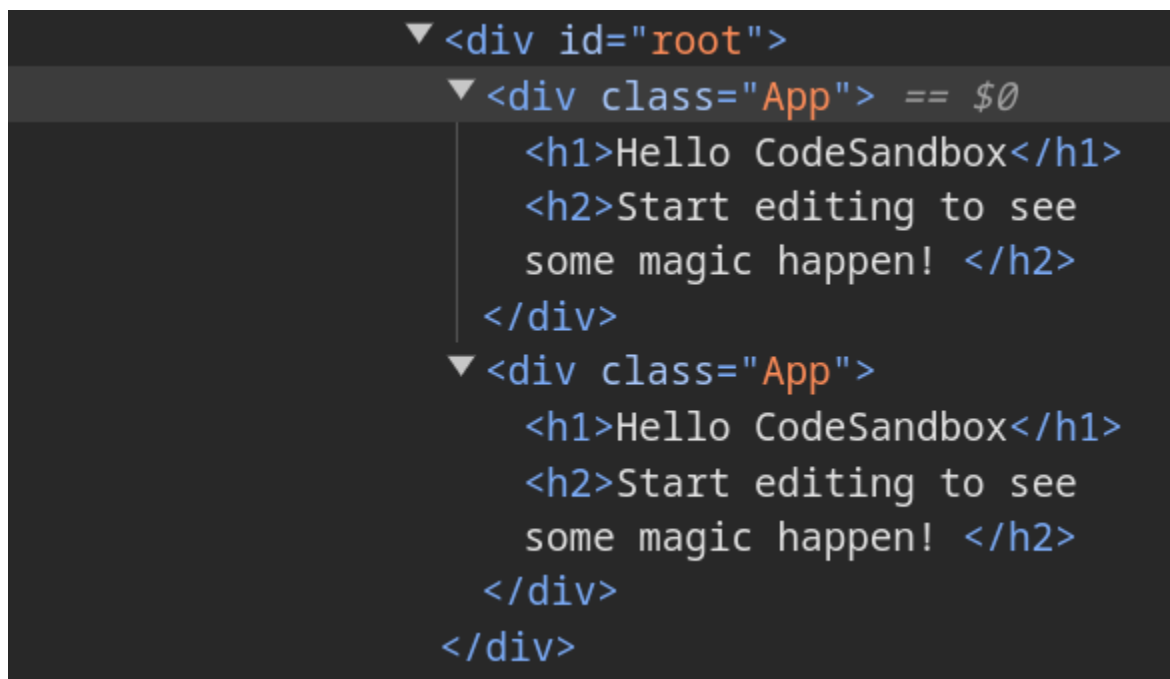
Let's use the shorthand syntax in our code.

```
import "./styles.css";

export default function App() {
  const greeting = "Hello CodeSandbox";
  const message = "Start editing to see some magic happen! ";
  return (
    <>
      <div className="App">
        <h1>{greeting}</h1>
        <h2>{message}</h2>
      </div>
      <div className="App">
        <h1>{greeting}</h1>
      </div>
    </>
  );
}
```

```
    <h2>{message}</h2>
  </div>
</>
);
}
```

Now the errors disappear and when we inspect the elements using browser tools, we can see something like this;



```
▼ <div id="root">
  ▼ <div class="App"> == $0
    <h1>Hello CodeSandbox</h1>
    <h2>Start editing to see
    some magic happen! </h2>
  </div>
  ▼ <div class="App">
    <h1>Hello CodeSandbox</h1>
    <h2>Start editing to see
    some magic happen! </h2>
  </div>
</div>
```

Components

React is a component-based framework. A component is a function that returns React elements. Components make up the UI of any React Application. A component can be as small as a button or as big as a **Navbar** or **Main** content.

React comes with two kinds of components, class-based components and **function-based** components. **Class-based** components are considered legacy, and it's recommended to use function-based components.

You can think of a component as a JavaScript function that returns **HTML**.

A single component contains content, styles, and javascript of that feature.

We already have the App component.

Create Your First Component

The first step to creating interactive user interfaces in React is to create **components** since they are the building blocks of any React application. Components can be created as standalone or inside other components. For example, in the **App.js** file, we can add more components to it.

To create a standalone file, you need to create a new file in the src directory. When naming components, ensure you use **PascalCasing**.

Create a new file named **Menu.js** . In the new component, let's define the structure and what it should display.

```
export default function Menu() {  
  return (  
    <div>  
      <h2>Hello there what are you having</h2>  
    </div>  
  );  
}
```

To make this component part of your app, let's import it into the **App.js** file at the top as follows:

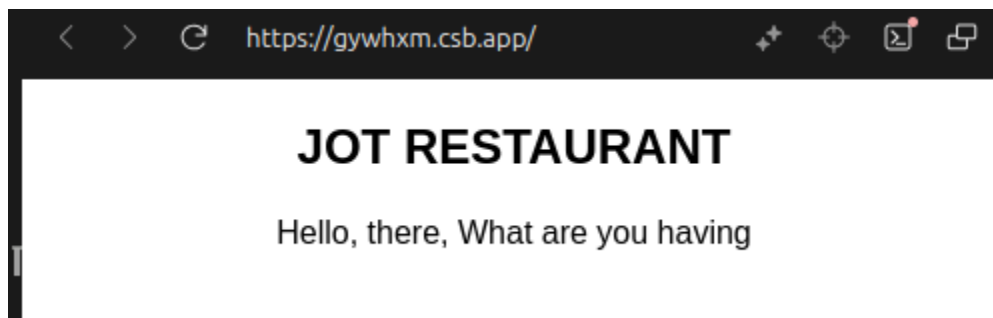
To render the Menu component as part of our UI, we need to render it as **<Menu/>** .

Update the App.js file as shown below.

```
import "./styles.css";  
import Menu from "./Menu";
```

```
export default function App() {
  return (
    <div className="App">
      <h1>JOT RESTAURANT</h1>
      <Menu />
    </div>
  );
}
```

Now your app looks like this.



The good thing about React is that, since it uses a **virtual DOM**, (a lightweight in-memory representation of the real DOM) every time you make changes to your code, the UI updates quickly without any page reload

Styling in React

You have probably seen the `import './styles.css';` import at the top of the App.js file. This is responsible for importing all the **CSS** styles to be applied to the App component elements.

React also allows us to add inline styles just as we would in HTML. However, the way we do it in React is a bit different. Instead of strings, React expects the styles to be contained in an object for the style attribute, with the CSS properties written in **camelCase**. Here's an example:


```
const styles = {
  color: 'blue',
  fontSize: '20px',
  marginTop: '10px'
};

function MyComponent() {
  return <div style={styles}>This is a styled text!</div>;
}
```

Notice how **font-size** becomes **fontSize**, and values are written as strings or numbers where appropriate.

Let's add some styles to our application .

```
import "./styles.css";

const styles = {
  color: "blue",
  fontSize: "20px",
  marginTop: "10px",
};

export default function App() {
  return (
    <div className="App">
      <h2 style={styles}>JOT RESTAURANT</h2>
    </div>

  );
}
```

These styles will be added to the **h2** tag and you will see that the heading has a blue color

Props

You know how functions in **JavaScript** have arguments, and React components have **props**. **Props** allow us to pass data from one component to another, typically from a parent component to a child component.

In React, data can only be passed from parent to child components, not the other way around, for example, we have a **Menu** and an **App** component, and since the App is the parent component, we can only pass data downwards to its children.

Props are also read-only and shouldn't be modified within a component. If a child component needs to communicate changes or pass data back up to the parent, it can do so using callback functions passed as **props**.

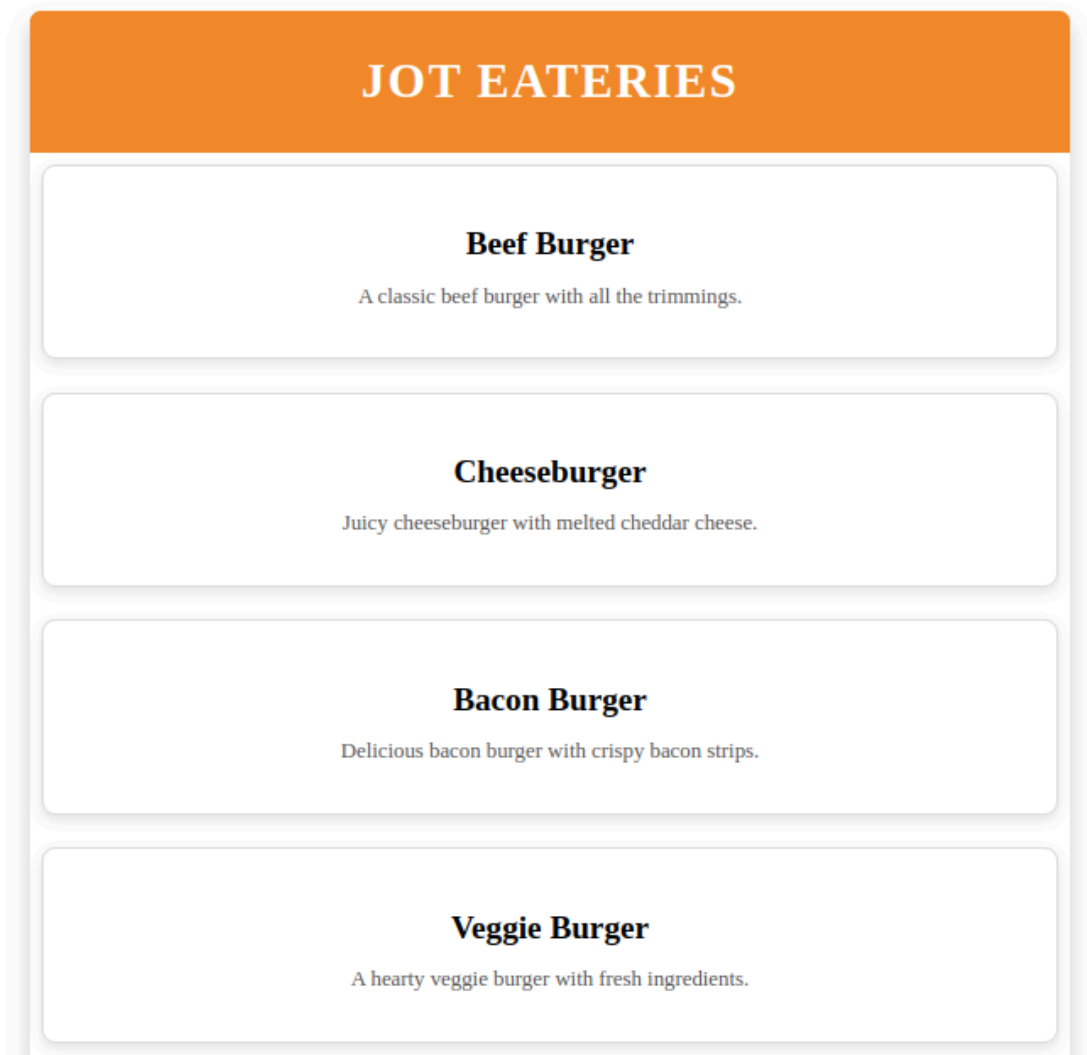
In the Menu component we want to add a few dishes, update like this:

```
// menu.js
import React from "react";

const Menu = () => {
  return (
    <div className="menu-container">
      <div className="burger-item">
        <h3>Beef Burger</h3>
        <p>A classic beef burger with all the trimmings.</p>
      </div>
      <div className="burger-item">
        <h3>Cheeseburger</h3>
        <p>Juicy cheeseburger with melted cheddar cheese.</p>
      </div>
      <div className="burger-item">
        <h3>Bacon Burger</h3>
        <p>Delicious bacon burger with crispy bacon strips.</p>
      </div>
      <div className="burger-item">
        <h3>Veggie Burger</h3>
        <p>A hearty veggie burger with fresh ingredients.</p>
      </div>
    </div>
  );
}
```

```
<div className="burger-item">
  <h3>Double Patty Burger</h3>
  <p>A massive burger with two juicy patties.</p>
</div>
<div className="burger-item">
  <h3>Spicy Jalapeño Burger</h3>
  <p>A spicy burger with jalapeños and pepper jack cheese.</p>
</div>
</div>
);
};
```

Our app now looks like this:



In the **Menu** component, we have at least 5 items displayed, each item is contained in its div and the div has the **title** and **description**. If we had images, we would also add them there. But there is a problem, this data is static. If we had a list of 100 items to display, it would be very cumbersome to create a div for each item, and that's where props come to save the day

Props (properties) allow us to pass data from parent to child components. We can use props to pass data therefore making components more reusable and dynamic.

Create a new **component** called **Menu** . Inside this file, we will create an instance of one burger, ie one item.

```
export default function Burger(props) {
  return (
    <div>
      <div className="burger-item">
        <h3>Beef Burger</h3>
        <p>A classic beef burger with all the trimmings.</p>
      </div>
    </div>
  );
}
```

You can see we have added the **props** in the function definition. Now rather than hardcoding the name of the burger and the description, we will simply use **props.name** and **props.description**

```
export default function Burger(props) {
  return (
    <div>
      <div className="burger-item">
        <h3>{props.name}</h3>
        <p>{props.description}</p>
      </div>
    </div>
  );
}
```

Now in the Menu component, we need to pass as many **Burger** instances as there are burgers, so if we have 5 burgers in the menu, we will pass 5 Burger component instances like this:

Let's start by adding one instance of the **Burger** component. First import the **Burger** component at the top of the Menu.js file.

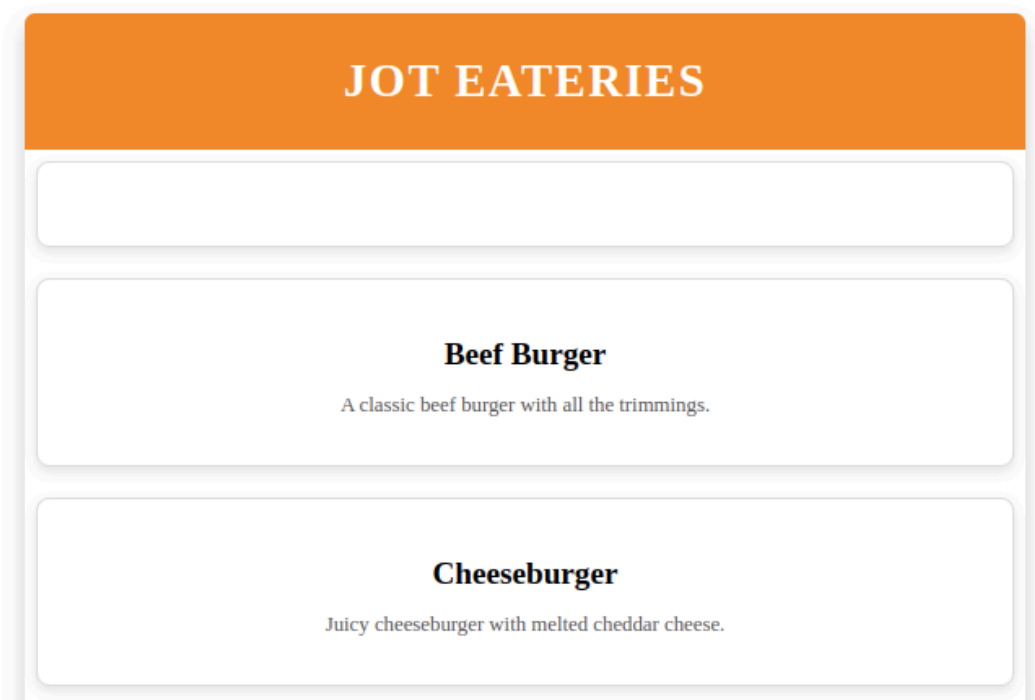
```
import Burger from "../Burger";
```

Update the **Menu.js** file as shown below:

```
// menu.js
import Burger from "./Burger";
export default function Menu() {

  return (
    <div className="menu-container">
      <Burger />
      {/* the rest of the code */}
    </div>
  );
};
```

The app now looks like this;



As you can see the first item is empty, that's because even though we have injected an instance of the **Burger** component, we haven't defined the values, the **Burger** component expects a name and a description.

This is how data is added:

```
<Burger name = "Beef Burger" description = "A classic beef burger with all the trimmings."/>
```

Now we don't need the hard-coded data, since we have a reusable component. let's replace the rest of the data

```
const Menu = () => {  
  return (  
    <div className="menu-container">  
      <Burger  
        name="Beef Burger"  
        description="A classic beef burger with all the trimmings."  
      />  
      <Burger  
        name="Cheeseburger"  
        description="Juicy cheeseburger with melted cheddar cheese."  
      />  
      <Burger  
        name="Bacon Burger"  
        description="Delicious bacon burger with crispy bacon  
strips."  
      />  
      <Burger  
        name="Veggie Burger"  
        description="A hearty veggie burger with fresh ingredients."  
      />  
      <Burger  
        name="Double Patty Burger"  
        description="A massive burger with two juicy patties."  
      />  
      <Burger  
        name="Spicy Jalapeño Burger"  
        description="A spicy burger with jalapeños and pepper jack  
cheese."  
      />  
    </div>  
  );  
};
```

```
};
```

```
}
```

We have successfully passed **props** from parent to child.

Destructuring Props

When we define **props** as we have done above, we have no idea of what the props object contains. Luckily with destructuring, we can destructure the properties **{name, description}** and then rather than referencing **props.name** or **props.description**, we do something like this;

```
export default function Burger({name,description}) {  
  return (  
    <div>  
      <div className="burger-item">  
        <h3>{name}</h3>  
        <p>{description}</p>  
      </div>  
    </div>  
  );  
}
```

List Items

If we were getting a large amount of data from an **API** or database, it would take us all day to directly add the burger components in the **JSX**. To improve reusability and maintainability, the best approach would be to define an array of burgers, and then iterate over the array and create a component for each burger.

Create an array named **burgers** and add all the burger names. The array should be defined at the top level, outside the function.

```
const burgers = [
```



```
{ name: "Beef Burger" },
{ name: "Cheeseburger" },
{ name: "Bacon Burger" },
{ name: "Veggie Burger" },
{ name: "Double Patty Burger" },
{ name: "Spicy Jalapeño Burger" },
{ name: "BBQ Burger" },
{ name: "Mushroom Swiss Burger" },
{ name: "Chicken Burger" },
{ name: "Fish Burger" },
{ name: "Impossible Burger" },
];
```

Keeping the array outside the component ensures the data is only created once when the module is loaded. It is also good for performance since it can lead to unnecessary performance overhead especially if the dataset is large., if we keep it inside the component, it means that it will be rendered every time the component mounts.

Keeping the data outside the **component** also prevents redundant initializations since the data is static.

Iterating with the Map() Method

We have worked with a small set of data so far, suppose that our data is coming from an API or a database, the same approach we did so far will not be suitable. The good thing about iterating in **React** is that you use pure JavaScript, so your Javascript skills of mapping over an array with the **map ()** method will come in handy

To recap, if we wanted to map over an array, we would do something like this:

```
const burgers = ["Beef Burger", "Cheeseburger", "Bac Burger"];

const burgerList = burgers.map((burger) => {
  return `<div>
    <h3>${burger}</h3>
  `;
});
```

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

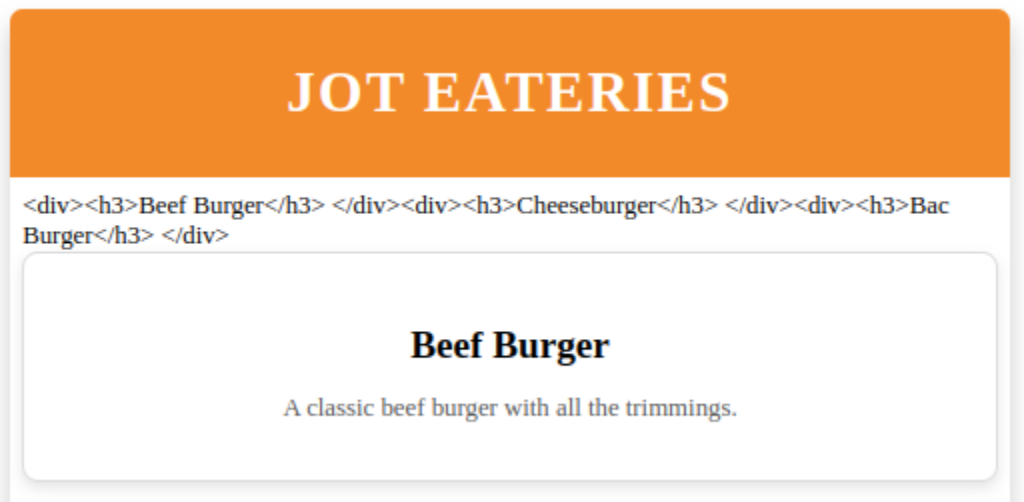
The **burgerList** array will now contain an array of HTML strings for each burger. The output will contain an array of HTML strings for each burger.

```
[  
  '<div><h3>Beef Burger</h3> </div>',  
  '<div><h3>Cheeseburger</h3> </div>',  
  '<div><h3>Bac Burger</h3> </div>'  
]
```

Luckily react does a good job of unpacking arrays, So for example, if we injected this data into our JSX syntax like this:

```
{  
  [  
    '<div><h3>Beef Burger</h3> </div>',  
    '<div><h3>Cheeseburger</h3> </div>',  
    '<div><h3>Bac Burger</h3> </div>'  
  ]  
}
```

This would be the output



We can see that react renders the data as required, all we need to do is apply the necessary styles.

Create an array of burger objects

```
const burgers = [
  {
    name: "Beef Burger",
    description: "A classic beef burger with all the trimmings.",
  },
  {
    name: "Cheeseburger",
    description: "Juicy cheeseburger with melted cheddar cheese.",
  },
  {
    name: "Bacon Burger",
    description: "Delicious bacon burger with crispy bacon strips.",
  },
  {
    name: "Veggie Burger",
    description: "A hearty veggie burger with fresh ingredients.",
  },
  {
    name: "Double Patty Burger",
    description: "A massive burger with two juicy patties.",
  },
  {
    name: "Spicy Jalapeño Burger",
    description: "A spicy burger with jalapeños and pepper jack cheese.",
  },
];
```

Since the **map()** method must return something, in this case we want to create a **Burger** component for each of the items in the burgers array and then pass the name

and description. Inside the **menu-container div**, call `burgers.map` and return a **Burger** component for each element.

```
import React from "react";

const burgers = [
  {
    name: "Beef Burger",
    description: "A classic beef burger with all the trimmings.",
  },
  {
    name: "Cheeseburger",
    description: "Juicy cheeseburger with melted cheddar cheese.",
  },
  {
    name: "Bacon Burger",
    description: "Delicious bacon burger with crispy bacon strips.",
  },
  {
    name: "Veggie Burger",
    description: "A hearty veggie burger with fresh ingredients.",
  },
  {
    name: "Double Patty Burger",
    description: "A massive burger with two juicy patties.",
  },
  {
    name: "Spicy Jalapeño Burger",
    description: "A spicy burger with jalapeños and pepper jack
cheese.",
  },
];

const Menu = () => {
  return (
    <div className="menu-container">
      {burgers.map((burger) => (
        <Burger name={burger.name} description={burger.description}
      />
```

```
    )})  
  </div>  
  );  
};  
  
export default Menu;
```

Once you update this, we can see an error that looks like this:

Error

Objects are not valid as a React child (found: object with keys {name}). If you meant to render a collection of children, use an array instead.

This error occurs because react needs each item in a list to have a unique **key**. To resolve the error, we need to add a key to each element. Since the **index** is unique, let's add it as the key. Although using the index as a key is not recommended because it can lead to issues especially if any of the list items is removed or shifted in position you might want to generate a unique key for each burger.

```
<div className="menu-container">  
  {burgers.map((burger,index) => (  
    <Burger key = {index}name={burger.name}  
description={burger.description} />  
  ))}  
</div>
```

Conditional Rendering

Conditional rendering lets you render data based on certain conditions, for example, suppose we wanted to render certain burgers only if they are available or spicy, let's add

additional properties to each burger namely **is_available(boolean)** and **isSpicy(boolean)**.

```
const burgers = [
  { name: "Beef Burger", description: "A classic beef burger with all the trimmings.", available: true, isSpicy: false },
  { name: "Cheeseburger", description: "Juicy cheeseburger with melted cheddar cheese.", available: true, isSpicy: false },
  { name: "Bacon Burger", description: "Delicious bacon burger with crispy bacon strips.", available: false, isSpicy: false },
  { name: "Veggie Burger", description: "A hearty veggie burger with fresh ingredients.", available: true, isSpicy: false },
  { name: "Double Patty Burger", description: "A massive burger with two juicy patties.", available: true, isSpicy: false },
  { name: "Spicy Jalapeño Burger", description: "A spicy burger with jalapeños and pepper jack cheese.", available: true, isSpicy: true }
];
```

Now we can conditionally render items based on these properties. This time, we will use the filter method to render only the burgers that are **spicy**.

```
const availableBurgers = burgers.filter((burger) =>
  burger.available);
```

In the code above, `.filter(burger => burger.isSpicy)` ensures that only burgers where `available: true` are passed to the `.map()` function for rendering.

Let's add the spicy Badge based on the **isSpicy** component within the burger component.

```
export default function Burger({ name, description, isSpicy }) {
  return (
    <div>
      <div className="burger-item">
        <h3>{name}</h3>
        <p>{description}</p>
```

```

        {isSpicy && <span className="spicy-badge">Spicy 🌶️</span>}
      </div>
    </div>
  );
}

```

`{isSpicy && Spicy 🌶️}` This code is one of the ways of applying conditional logic in React. This essentially means if `isSpicy` is true, render a span element.

Lastly, update the `map` method in the `Menu` component to use the `available` array rather than the `burgers` array.

```

const Menu = () => {
  return (
    <div className="menu-container">
      {availableBurgers.map((burger, index) => (
        <Burger
          key={index}
          name={burger.name}
          description={burger.description}
          isSpicy={burger.isSpicy}
        />
      ))}
    </div>
  );
};

```

Now your app looks like this:

JOT EATERIES

Beef Burger

A classic beef burger with all the trimmings.

Spicy 🌶️

Cheeseburger

Juicy cheeseburger with melted cheddar cheese.

Spicy 🌶️

Veggie Burger

A hearty veggie burger with fresh ingredients.

We can see that the spicy badge has been added to the spicy burgers and now we are only showing the available burgers.

Other examples of Conditional Rendering

If the burgers array didn't have any data, we would have a blank page and this is not a good user experience. To prevent that we can create a condition that checks if the length of the burgers array is equal to 0 and if it is, we just display a message like “ **No burgers available at the moment. Please check back later!**”


```

const Menu = () => {
  if (availableBurgers.length === 0) {
    return <p>No burgers available at the moment. Please check back
later!</p>;
  }
  return (
    <div className="menu-container">
      {availableBurgers.map((burger, index) => (
        <Burger
          key={index}
          name={burger.name}
          description={burger.description}
          isSpicy={burger.isSpicy}
        />
      ))}
    </div>
  );
};

```

Even though **JSX** looks similar to **HTML** markup, there are some key differences you will encounter when working with different elements. For example, let's look at the input and image elements.

```

<>

  
  <input type="text" placeholder="Type something..." />

</>

```

JSX Tags must be Closed

Both `` and `<input>` are examples of self-closing tags. In HTML, it is okay to leave self-closing tags open e.g (``), however in JSX, you must explicitly close them by adding a slash (/) before the closing `>`.

If you don't close these elements, you will get an error like this:



```
// Incorrect:  
<img>
```

```
// Correct:  
<img />
```

JSX Attributes must use CamelCase

When you need to respond to an event on a button, you typically pass the function as shown below for an **onclick** event.

```
<button onclick="handleClick()">Click me</button>
```

In HTML, **onclick** is written in all lowercase. When a user clicks the button, the **handleClick** function will be called.

However, in JSX, attributes that are Multi-word or part of the JavaScript language must be written in **camelCase**. So the equivalent JSX will look like this:

```
function handleClick() {  
  alert("Hello");  
}  
  
export default function App() {  
  return (  
    <button onClick={() => handleClick("Hello")}>Click me</button>  
  );  
}
```

In the code above, **onClick** is written in **camelCase** instead of onclick, and the function handleClick is wrapped in curly braces to indicate that it is a JavaScript expression.

You might also have noticed that the function handleClick has not been invoked when passed to the **onClick** attribute. This is because when the component renders, we don't want to invoke the function immediately. In JSX, when handling event handlers, the function should be passed as a reference without parentheses to avoid immediate invocation on component mount.

What if you need to pass arguments to the function? In this case, you use an arrow function which will call the handler with the desired arguments.

```
function handleClick(message) {  
  alert(message);  
}  
  
export default function App() {  
  return (  
    <button onClick={() => handleClick("Button clicked!")}>Click  
me</button>  
  );  
}
```

```
}
```

Conditionals in JSX

In JSX, conditional statements like `if`, `for`, `switch`, etc. are not allowed directly inside **JSX** markup. If you need to perform conditional rendering, you can use ternary operators or logical operators to conditionally render based on certain conditions.

```
// Incorrect:
if (isLoggedIn) {
  return <h1>Welcome</h1>;
}

// Correct (using ternary operator):
{isLoggedIn ? <h1>Welcome</h1> : <h1>Please Log In</h1>}
```

State Management in React

In this section, we will learn about **state** and how **React** uses it to update all areas of the application."

We will start with a simple **counter** example. The purpose of this counter is to increment and decrement numbers when a button is clicked.

Create a new React application and in the **App.js** file, add the code below.

```
import './styles.css';
import { useState } from 'react';
```

```
export default function App() {
  const [count, setCount] = useState(0);

  return (
    <div className="counter">
      <button>-</button>
      <span className="counter-number">{count}</span>
      <button>+</button>
    </div>
  );
}
```

The counter now looks like this:



In **React**, state is used to manage dynamic content. To create a state variable, we use the **useState** function. Whose syntax looks like this:

```
const [value, setValue] = useState(initialValue)
```

Where **value** is the current state, **setValue** is the function used to update the **state**, and **initialValue** is the initial value of the state. The initial value can be empty, an array, an object, or a string.

In our example, **count** is the current value of the state, and its initial value is 0. **setCount** is the function used to update the state.

To update the counter, we need to add **onClick** event handlers to the buttons. The left button will call a function to decrease the **counter**, while the right button will call a function to increase the counter by 1.

To update the new state when a button is clicked, you can do so directly by calling the setter function as follows:

```
setValue(value)
```

Use the State in JSX

Let's update the state directly within our button using the **setCount** function within the **onClick** event handler.

```
<div className="counter">
  <button onClick={() => setCount(count - 1)}>-</button>
  <span className="counter-number">{count}</span>
  <button onClick={() => setCount(count + 1)}>+</button>
</div>
```

When you click any of the buttons, the counter will change. Under the hood, when the state changes, it causes the component to re-render. Even though you don't see the page refresh, React does a good job of updating only the parts of the DOM that need to change.

State management in Forms

Working with **forms** can be a challenging process. You need to create the correct inputs for the **data**, perform validation on both the front end and back end, and ensure that the submitted data is valid and adheres to the required format and length.

In a traditional **HTML** form, data is typically collected from the input fields after the user clicks the submit button. However, working with forms in React is a bit different and slightly more complex. In React, every change in the input data must be captured and maintained in the component's **state**.

By the time the user clicks the submit button, your application already has the input data stored in its state.

This approach offers some benefits mainly:

- **Immediate feedback:** Since the **state** handles the submitted data, validation can be done immediately before the user even submits the form.
- **State controlled:** In React, we often hear the terms controlled vs uncontrolled input. In controlled input, as the name suggests, the input value is derived from the component's **state** meaning that every change in the input updates the state.

A simple input markup looks like this:

```
<input type="text" id="name" name="name" value="John">
```

In this traditional **HTML** form, we have the value "John," which has been hardcoded. However, in the **React** context, inputs are controlled elements, meaning that their values are managed by the state. Therefore, we will bind the input value to a **state** variable and handle changes to the value through state.

In **React**, the input will look like this:

```
<input type="text" id="name" name="name" value={name} onChange={(e) => setName(e.target.value)} />
```

Assuming we have a name state

```
const [name, setName] = useState()
```

Let's create a simple **Login** form with **username** and **password** fields in the App.js file.

```
import "./styles.css";

export default function App() {
  return (
    <div className="login-container">
      <form className="login-form">
        <h2>Login</h2>
        <div className="input-group">
          <label htmlFor="username">Username</label>
          <input type="text" id="username" name="username" />
        </div>
        <div className="input-group">
          <label htmlFor="password">Password</label>
          <input type="password" id="password" name="password" />
        </div>
        <button type="submit" className="login-button">
          Login
        </button>
      </form>
    </div>
  );
}
```

How State Works in Forms

In React, the value attribute of an input field is controlled by the component's **state**. This means that whatever is in the initial state will be the value shown in the input field. For example, let's create a state for both of our input fields:

First, import the **useState** function at the top of the file.

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

Initialise state variables for username and password.

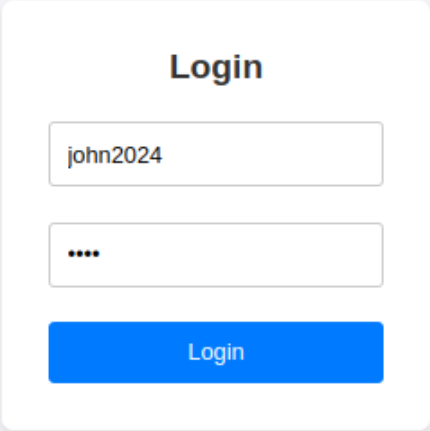
```
const [username, setUsername] = useState("john2024")
const [password, setPassword] = useState("1234")
```


Bind the value to the state

```
<input type="text" id="username" name="username"
value = {username} placeholder = "username"/>

<input type="password" id="password" name="password"
value = {password}
placeholder = "password"/>
```

Once we bind the value to the **state**, we can see that the values shown on the input fields are from the state.



The image shows a login form with a white background and rounded corners, centered on a light purple background. The form has a title 'Login' in bold black text. Below the title are two input fields: the first is a text field containing 'john2024', and the second is a password field with four dots. Below the input fields is a blue button with the text 'Login' in white.

The next step is to update the **state** when the input value changes. This is done by adding the **onChange** event handler which listens for changes to the input field. As the user types in the input, the event is triggered and the state is updated by the **setter** function.

This update happens in real-time and is reflected in the input field

Let's add **onChange** handlers to both our input fields:

```
<input
  type="text"
  id="username"
```

```

    name="username"
    value={username}
    placeholder="username"
    onChange={(e) => setUsername(e.target.value)}
  />

  <input
    type="password"
    id="password"
    name="password"
    value={password}
    placeholder="password"
  />

```

It's recommended to set the initial value of the **state** for an input field to an empty string, so let's do that.

```

const [username, setUsername] = useState(" ");
const [password, setPassword] = useState(" ");

```

Now the input fields are controlled by the **state**, and we are in a position to submit form data. Let's do that. Add an **onClick** event handler to the submit button.

```

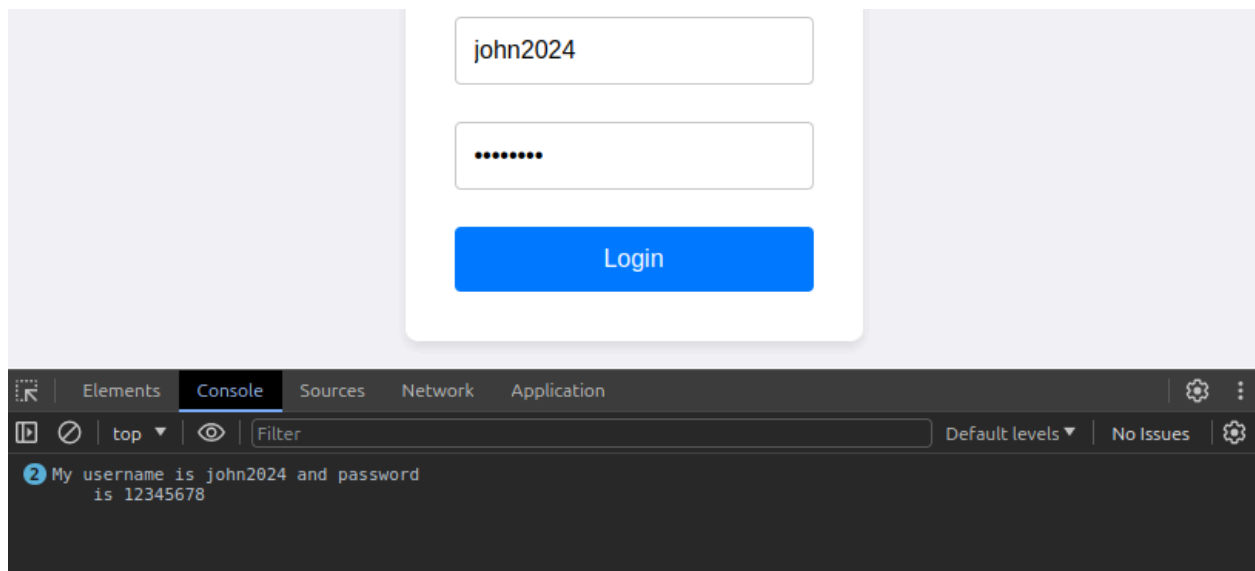
<button onClick={submitData} type="submit" className="login-button">
  Login
</button>

```

Create the submitData function and log the form data to the console.

```
function submitData(e) {  
  e.preventDefault();  
  // send data to server  
  
  console.log(`My username is ${username} and password  
  is ${password} `);  
}
```

Here is the output:



💡 In React, it's important to understand that state updates are asynchronous and are typically batched together for performance optimization.

Lifting State in React

Since **React** is a component-based framework, it's common to create independent components that use their own state. However, in many cases, multiple components need to share or synchronize data. When this happens, the state must be **lifted up** to a common parent component so that it can be passed down as props to child components. This process is known as **lifting state**.

An example that comes to mind is a **Modal**, If we create an independent modal, but the button for opening the modal lives in the parent component, we have to lift the **state** on the modal into the parent component so as to effectively manage it.

Suppose we have a **Modal** component that looks like this:

```
import React, { useState } from "react";
import styles from "./Modal.module.css";
export default function Modal(props) {
  const [isOpen, setIsOpen] = useState(false);

  const handleOpen = () => {
    setIsOpen(true);
  };

  const closeModal = () => {
    setIsOpen(false);
  };

  return (
    <>
      <button onClick={handleOpen}>Open Modal</button>
      <div>
        {isOpen && (
          <div className={styles.overlay}>
            <div className={styles.modal}>
              <h2 className={styles.title}>Simple Modal</h2>
              <p className={styles.content}>
                This is a simple modal component </p>
              <button onClick={closeModal}
                className={styles.closeButton}>
                Close Modal
              </button>
            </div>
          </div>
        )}
      </div>
    </div>
  )
}
```

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

In this code, we have a state that controls the opening and closing of a **modal**. When **isOpen** is true, the **modal** is shown; if it is false, the modal is not displayed. However, we have a problem: we want the button for opening the modal to reside in the App component.

The App component cannot change the state of the modal since it has no access to the **state**. In such a case, we have to lift the state from the Modal component and put it in the App component.

This will allow us to add an **OnClick** event which will trigger a state change.

Let's start by moving the **isOpen** state and the functions that control the modal to the App component.

```
function App() {
  const [isOpen, setIsOpen] = useState(false);
  const handleOpen = () => {
    setIsOpen(true);
  };

  const closeModal = () => {
    setIsOpen(false);
  };

  return (
    <>
      <button >Open Modal</button>
      <Modal isOpen={isOpen} closeModal={closeModal} />
    </>
  );
}
```

The next step is to modify the **Modal** component so that it receives the state (**isOpen**) and the **closeModal** function from the App component as props.

```
export default function Modal({isOpen, closeModal }) {  
  // const [isOpen, setIsOpen] = useState(false);  
  
  return (  
    <>  
      {/* <button onClick={props.handleOpen}>Open Modal</button> */}  
      <div>  
        {isOpen && (  
          <div className={styles.overlay}>  
            <div className={styles.modal}>  
              <h2 className={styles.title}>Simple Modal</h2>  
              <p className={styles.content}>  
                This is a simple modal component.  
              </p>  
              <button onClick={closeModal}  
className={styles.closeButton}>  
                Close Modal  
              </button>  
            </div>  
          </div>  
        )}  
      </div>  
    </>  
  );  
}
```

Lastly, attach an event listener to the open **Modal** function so that it will be responsible for opening the modal.

```
<button onClick={handleOpen}>Open Modal</button>
```

Now that the App component controls the state of the Modal, and is also the single source of truth for the state of the Modal, it allows us to share the state with multiple child elements and also control the state from different parts of our application.

Managing Side Effects with the useEffect Hook

Side effects in **React** are operations that affect other components and for which **React's** built-in rendering logic can't handle on its own. Some common examples these side effects include

- **API requests**
- **Local storage**
- **Event listeners**
- **Subscriptions such as web sockets, e.t. c**

Let's look at a few examples.

API Requests

When you need to fetch data from an API in React, it creates a side effect since it involves **asynchronous** operations. Since React components render on mount, an asynchronous operation may not be completed in time before the component's initial render. In such cases, we use the **useEffect** hook to perform API interactions.

Suppose you need to fetch data from an **API**, such as a placeholder that returns some mock data. Let's create a **useEffect** for that. In your App.js file, start by importing the **useEffect** hook."

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

Next, define it

```
useEffect(()=>{  
  
  //put side effect code here  
})
```

So for example in our case, we need to do a **fetch** request, let's do that inside the **useEffect** hook.

```
function App() {  
  
  useEffect(() => {  
    fetch('https://jsonplaceholder.typicode.com/todos/1')  
      .then((res) => res.json())  
      .then((data) => {  
        console.log(data);  
      });  
  });  
}
```

Now , everytime the app renders, the data will be fetched and logged to the console.

The dependency array

React offers another argument to the **useEffect** hook, which is the dependency array. The dependency array determines what causes the **useEffect** to run. If you don't want the **useEffect** to run on every re-render, you can specify the dependency array.

For example, if we want the **useEffect** to run on component mount, we will specify an empty dependency array, as shown below

```
useEffect(() => {  
  fetch('https://jsonplaceholder.typicode.com/todos/1')  
    .then((res) => res.json())  
    .then((data) => {  
      console.log(data);  
    });  
}, []);
```


If you want the use **Effect** to run when something like **state** changes, you will add the state to the dependency array,

For example, if we have a user state and we want it to control when the **useEffect** runs, we would have something like this:

```
function App() {  
  
  const [user, setUser] = useState({ "name": "Ankit", "age": 25 });  
  
  useEffect(() => {  
    fetch('https://jsonplaceholder.typicode.com/todos/1')  
      .then((res) => res.json())  
      .then((data) => {  
        console.log(data);  
      });  
  }, [user]);  
}
```

Local Storage

Reading and writing to **local storage** is a side effect and should be handled in a similar manner using the **useEffect** hook.

Consider this example where we have a simple note-taking app where a user adds notes, and the notes are then displayed on the page

```
function App() {  
  const [note, setNote] = useState("");  
  const [notes, setNotes] = useState([]);  
  console.log(notes);  
}
```

```

return (
  <>
    <input type="text" value={note}
onChange={ (e)=>setNote(e.target.value)}/>
    <button onClick={ ()=>setNotes([...notes,note])}>Add
Note</button>

    {notes? notes.map((note,i)=>{
      return <div key={i}>{note}</div>
    }):null}

  </>
);
}

```

When a user refreshes the app, all the notes will disappear. To ensure that all the notes are displayed even after a refresh, we will store the notes in **local storage**.

Add a **useEffect** hook to save the current notes to **local storage** whenever the note state changes. In this case, our dependency array will be `note`. This means that every time a user adds a new note, the note will be saved to **local storage**.

```

useEffect(()=>{
  localStorage.setItem("notes",JSON.stringify(notes));
},[notes])

```

Then we need to set the initial state of the notes to be the notes from **local storage**.

```

const[notes,setNotes]=useState(localStorage.getItem("notes"?
JSON.parse(localStorage.getItem("notes")):[]);

```

Here we are using a ternary operator which will first check if the item `notes` exists in local storage and if it does, the array will be used as the initial value, if it doesn't exist, the **initial state** will be an empty array.

Here is the full code for managing side effects in React

```

import { useState,useEffect } from "react";
import "./App.css";

function App() {
  const [note,setNote]=useState("");
  const [notes,setNotes]=useState(localStorage.getItem("notes"?
JSON.parse(localStorage.getItem("notes")):[]);

  useEffect(() => {
    fetch('https://jsonplaceholder.typicode.com/todos/1')
      .then((res) => res.json())
      .then((data) => {
        console.log(data);
      });
  },[]);

  useEffect(()=>{
    localStorage.setItem("notes",JSON.stringify(notes));
  },[notes])
  function saveNote() {
    if (note.trim()) {
      setNotes((prevNotes) => [...prevNotes, note]);
      setNote("");
    }
  }
  return (
    <>

    <input type="text" value={note}
onChange={(e)=>setNote(e.target.value)}/>
    <button onClick={saveNote}>Add Note</button>

    {notes? notes.map((note,i)=>{
      return <div key={i}>{note}</div>
    }):null}
    </>
  );
}

```

```
}
```

Conclusion

Congratulations! You've just completed an introductory journey into the world of React. Let's recap what we've learned:

1. React basics and its component-based architecture
2. JSX syntax and how it differs from HTML
3. Creating and using components
4. Managing state with the `useState` hook
5. Handling user interactions and events
6. Conditional rendering and list rendering
7. Working with forms in React
8. Lifting state up for better component communication
9. Managing side effects with the `useEffect` hook

As you continue your React journey, practice is key. Try building small projects that incorporate these concepts, and don't be afraid to explore the React documentation for more in-depth information.