React

React is a JavaScript library for building user interfaces.

React is used to build single-page applications.

React allows us to create reusable UI components.

What is React?

React, sometimes referred to as a frontend JavaScript framework, is a JavaScript library created by Facebook.

React is a tool for building UI components.

How does React Work?

React creates a VIRTUAL DOM in memory.

Instead of manipulating the browser's DOM directly, React creates a virtual DOM in memory, where it does all the necessary manipulating, before making the changes in the browser DOM.

React only changes what needs to be changed!

React finds out what changes have been made, and changes **only** what needs to be changed.

You will learn the various aspects of how React does this in the rest of this tutorial.

React.JS History

Current version of React.JS is V18.0.0 (April 2022).

Initial Release to the Public (V0.3.0) was in July 2013.

React.JS was first used in 2011 for Facebook's Newsfeed feature.

Facebook Software Engineer, Jordan Walke, created it.

Current version of `create-react-app` is v5.0.1 (April 2022).

`create-react-app` includes built tools such as webpack, Babel, and ESLint.

To use React in production, you need npm which is included with Node.js.

To get an overview of what React is, you can write React code directly in HTML.

But in order to use React in production, you need npm
and Node.js installed.

React Directly in HTML

The quickest way start learning React is to write React directly in your HTML files.

Start by including three scripts, the first two let us write React code in our JavaScripts, and the third, Babel, allows us to write JSX syntax and ES6 in older browsers.

```
<!DOCTYPE html>
<html>
  <head>
    <script src="https://unpkg.com/react@18/umd/react.development.js"
crossorigin></script>
    <script src="https://unpkg.com/react-dom@18/umd/react-
dom.development.js" crossorigin></script>
    <script
src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
  </head>
  <body>

    <div id="mydiv"></div>

    <script type="text/babel">
```

```
      function Hello() {
        return <h1>Welcome to First React Example!</h1>;
      }

      const container = document.getElementById('mydiv');
      const root = ReactDOM.createRoot(container);
      root.render(<Hello />)
    </script>

  </body>
</html>
```

This way of using React can be OK for testing purposes, but for production you will need to set up a **React environment**.

Setting up a React Environment

If you have npx and Node.js installed, you can create a React application by using `create-react-app`.

If you've previously installed `create-react-app` globally, it is recommended that you uninstall the package to ensure npx always uses the latest version of `create-react-app`.

To uninstall, run this command: `npm uninstall -g create-react-app`.

Run this command to create a React application named `my-react-app`:

```
npx create-react-app my-react-app
```

The `create-react-app` will set up everything you need to run a React application.

Run the React Application

Now you are ready to run your first *real* React application!

Run this command to move to the `my-react-app` directory:

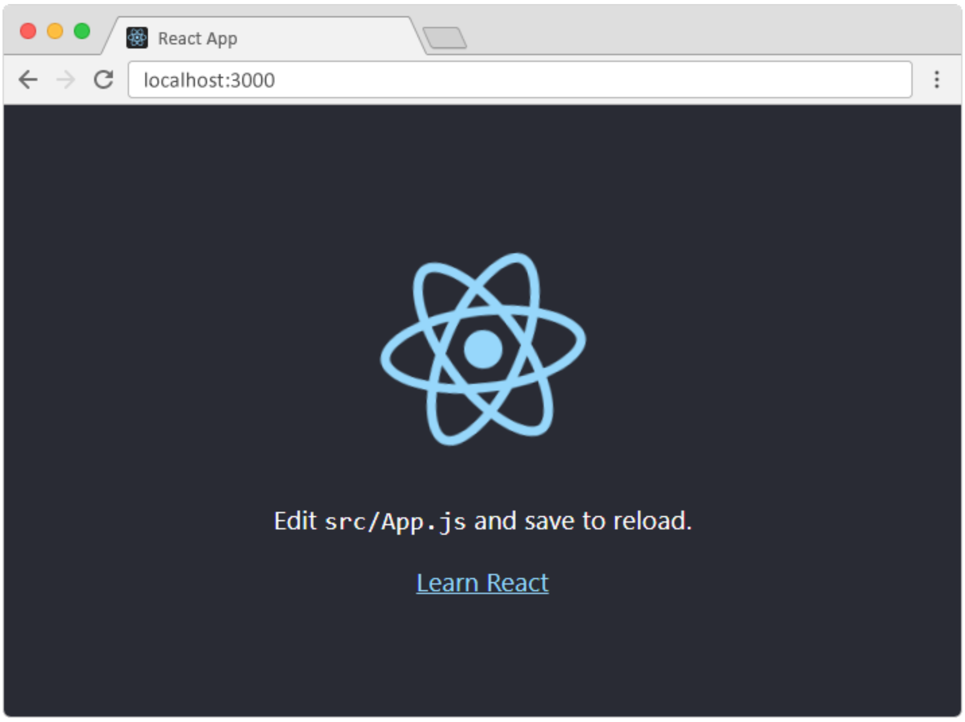
```
cd my-react-app
```

Run this command to run the React application `my-react-app`:

```
npm start
```

A new browser window will pop up with your newly created React App! If not, open your browser and type `localhost:3000` in the address bar.

The result:



Modify the React Application

So far so good, but how do I change the content?

Look in the `my-react-app` directory, and you will find a `src` folder. Inside the `src` folder there is a file called `App.js`, open it and it will look like this:

/myReactApp/src/App.js:

```
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;
```

Try changing the HTML content and save the file.

Notice that the changes are visible immediately after you save the file, you do not have to reload the browser!

Example

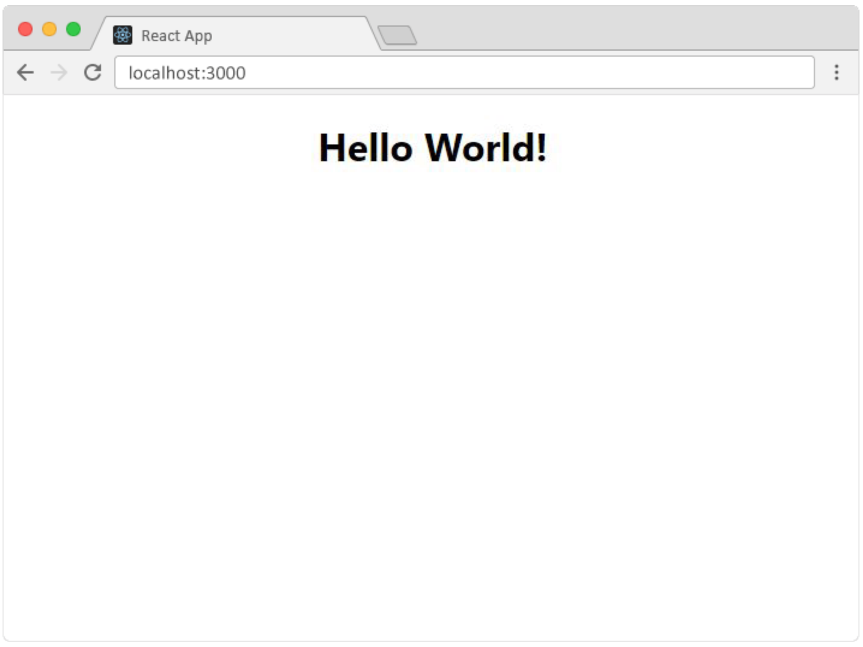
Replace all the content inside the `<div className="App">` with a `<h1>` element.

See the changes in the browser when you click Save.

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

  export default App;
```

Notice that we have removed the imports we do not need (logo.svg and App.css).

The result:



Hello World!

Next?

Now we have a React Environment on our computer.

If we want to follow the same steps on our computer, start by stripping down the `src` folder to only contain one file: `index.js`. We should also remove any unnecessary lines of code inside the `index.js` file to make them look like the example in the "Show React" tool below:

Example

Click the "Run Example" button to see the result.

`index.js`:

```
import React from 'react';

import ReactDOM from 'react-dom/client';



const myFirstElement = <h1>Hello React!</h1>



const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(myFirstElement);
```

React ES6

What is ES6?

ES6 stands for ECMAScript 6.

ECMAScript was created to standardize JavaScript, and ES6 is the 6th version of ECMAScript, it was published in 2015, and is also known as ECMAScript 2015.

Why Should I Learn ES6?

React uses ES6, and you should be familiar with some of the new features like:

- Classes
- Arrow Functions
- Variables (let, const, var)
- Array Methods like `.map()`
- Destructuring
- Modules
- Ternary Operator
- Spread Operator

Classes

ES6 introduced classes.

A class is a type of function, but instead of using the keyword `function` to initiate it, we use the keyword `class`, and the properties are assigned inside a `constructor()` method.

Example

A simple class constructor:

```
class Car {
  constructor(name) {
    this.brand = name;
  }
}
```

Notice the case of the class name. We have begun the name, "Car", with an uppercase character. This is a standard naming convention for classes.

Now you can create objects using the Car class:

Example

Create an object called "mycar" based on the Car class:

```
class Car {
  constructor(name) {
    this.brand = name;
  }
}

const mycar = new Car("Ford");
```

Note: The constructor function is called automatically when the object is initialized.

```
<!DOCTYPE html>
<html>
<body>
<script>
class Car {
  constructor(name) {
    this.brand = name;
  }
}
const mycar = new Car("Ford");
document.write(mycar.brand);
</script>
</body>
</html>
```

Method in Classes

You can add your own methods in a class:

Example

Create a method named "present":

```
class Car {
  constructor(name) {
    this.brand = name;
  }

  present() {
    return 'I have a ' + this.brand;
  }
}

const mycar = new Car("Ford");
mycar.present();
```

As you can see in the example above, you call the method by referring to the object's method name followed by parentheses (parameters would go inside the parentheses).

Class Inheritance

To create a class inheritance, use the `extends` keyword.

A class created with a class inheritance inherits all the methods from another class:

Example

Create a class named "Model" which will inherit the methods from the "Car" class:

```
class Car {

  constructor(name) {

    this.brand = name;

  }


  present() {

    return 'I have a ' + this.brand;

  }

}


class Model extends Car {

  constructor(name, mod) {

    super(name);

    this.model = mod;

  }
  show() {

      return this.present() + ', it is a ' + this.model

  }

}

const mycar = new Model("Ford", "Mustang");

mycar.show();
```

The `super()` method refers to the parent class.

By calling the `super()` method in the constructor method, we call the parent's constructor method and get access to the parent's properties and methods.

React ES6 Arrow Functions

Arrow Functions

Arrow functions allow us to write shorter function syntax:

Example

Before:

```
hello = function() {

  return "Hello World!";

}
```

<!DOCTYPE html>

<html>

<body>

<h1>Function</h1>

<p>This demonstrates a regular function, NOT an arrow function.</p>

<p id="msg"></p>

<script>

hello = function() {

  return "Welcome to JSPM!";

}

document.getElementById("msg").innerHTML = hello();

</script>

</body>

</html>

Example

With Arrow Function:

```
hello = () => {

  return "Hello World!";

}
```

<!DOCTYPE html>

<html>

<body>

<h1>Arrow Function</h1>

<h2>Implicit Return</h2>

<p>The arrow function expects a return value, and returns the value by default, without the <strong>return</strong> keyword.</p>

<p id="demo"></p>

<script>

hello = () => "Hello World!";

document.getElementById("demo").innerHTML = hello();

</script>

</body>

</html>

It gets shorter! If the function has only one statement, and the statement returns a value, you can remove the brackets *and* the `return` keyword:

Example

Arrow Functions Return Value by Default:

```
hello = () => "Hello World!";
```

```
<!DOCTYPE html>

<html>

<body>

<h1>Arrow Function</h1>

<h2>Implicit Return</h2>

<p>The arrow function expects a return value, and returns the value by default, without the <strong>return</strong> keyword.</p>

<p id="demo"></p>

<script>

hello = () => "Hello World!";

document.getElementById("demo").innerHTML = hello();

</script>

</body>

</html>
```

Note: This works only if the function has only one statement.

If you have parameters, you pass them inside the parentheses:

Example

Arrow Function With Parameters:

```
hello = (val) => "Hello " + val;
```

```
<!DOCTYPE html>

<html>

<body>

<h1>Arrow Function</h1>

<p>A demonstration of an arrow function in one line, with parameters.</p>

<p id="msg"></p>

<script>

hello = (val) => "Welcome to " + val;

document.getElementById("msg").innerHTML = hello("JSPM!");

</script>

</body>

</html>
```

In fact, if you have only one parameter, you can skip the parentheses as well:

Example

Arrow Function Without Parentheses:

```
hello = val => "Hello " + val;
```

```
<!DOCTYPE html>
<html>
<body>

<h1>Arrow Function</h1>

<p>As you can see in this example, you can skip the parentheses when you have only one parameter.</p>
<p id="msg"></p>
<script>
hello = val => "Hi " + val;
document.getElementById("msg").innerHTML = hello("JSPM!");
</script>
```

</body>
</html>

React ES6 Variables

Variables

Before ES6 there was only one way of defining your variables: with the `var` keyword. If you did not define them, they would be assigned to the global object. Unless you were in strict mode, then you would get an error if your variables were undefined.

Now, with ES6, there are three ways of defining your variables: `var`, `let`, and `const`.

Example

var

```
var x = 5.6;
```

If you use `var` outside of a function, it belongs to the global scope.

If you use `var` inside of a function, it belongs to that function.

If you use `var` inside of a block, i.e. a for loop, the variable is still available outside of that block.

`var` has a *function* scope, not a *block* scope.

Example

let

```
let x = 5.6;
```

`let` is the block scoped version of `var`, and is limited to the block (or expression) where it is defined.

If you use `let` inside of a block, i.e. a for loop, the variable is only available inside of that loop.

`let` has a *block* scope.

Example

const

```
const x = 15.6;
```

`const` is a variable that once it has been created, its value can never change.

`const` has a *block* scope.

The keyword `const` is a bit misleading.

It does not define a constant value. It defines a constant reference to a value.

Because of this you can NOT:

- Reassign a constant value
- Reassign a constant array
- Reassign a constant object

  But you CAN:

- Change the elements of constant array
- Change the properties of constant object

React ES6 Array Methods

Array Methods

There are many JavaScript array methods.

One of the most useful in React is the `.map()` array method.

The `.map()` method allows you to run a function on each item in the array, returning a new array as the result.

In React, `map()` can be used to generate lists.

Example

Generate a list of items from an array:

```
const myArray = ['apple', 'banana', 'orange'];



const myList = myArray.map((item) => <p>{item}</p>)
```

```
import React from 'react';
import ReactDOM from 'react-dom/client';

const myArray = ['apple', 'banana', 'orange'];

const myList = myArray.map((item) => <p>{item}</p>)

const container = document.getElementById('root');
const root = ReactDOM.createRoot(container);
root.render(myList);
```

React ES6 Destructuring

Destructuring

To illustrate destructuring, we'll make a sandwich. Do you take everything out of the refrigerator to make your sandwich? No, you only take out the items you would like to use on your sandwich.

Destructuring is exactly the same. We may have an array or object that we are working with, but we only need some of the items contained in these.

Destructuring makes it easy to extract only what is needed.

Destructing Arrays

Here is the old way of assigning array items to a variable:

Example

Before:

```
const vehicles = ['mustang', 'f-150', 'expedition'];
```

```
// old way

const car = vehicles[0];

const truck = vehicles[1];

const suv = vehicles[2];
```

Here is the new way of assigning array items to a variable:

Example

With destructuring:

```
const vehicles = ['mustang', 'f-150', 'expedition'];



const [car, truck, suv] = vehicles;
```

When destructuring arrays, the order that variables are declared is important.

If we only want the car and suv we can simply leave out the truck but keep the comma:

```
const vehicles = ['mustang', 'f-150', 'expedition'];



const [car,, suv] = vehicles;
```

Destructuring comes in handy when a function returns an array:

Example

```
function calculate(a, b) {

  const add = a + b;

  const subtract = a - b;

  const multiply = a * b;

  const divide = a / b;



  return [add, subtract, multiply, divide];
}
const [add, subtract, multiply, divide] = calculate(4, 7);
```

```
<!DOCTYPE html>

<html>

<body>

<script>

function calculate(a, b) {

  const add = a + b;

  const subtract = a - b;

  const multiply = a * b;

  const divide = a / b;

  return [add, subtract, multiply, divide];

}

const [add, subtract, multiply, divide] = calculate(4, 7);

document.write("<p>Sum: " + add + "</p>");

document.write("<p>Difference " + subtract + "</p>");

document.write("<p>Product: " + multiply + "</p>");

document.write("<p>Quotient " + divide + "</p>");

</script>

</body>

</html>
```

Destructuring Objects

Here is the old way of using an object inside a function:

Example

Before:

```
const vehicleOne = {

  brand: 'Ford',

  model: 'Mustang',

  type: 'car',

  year: 2021,

  color: 'red'

}


myVehicle(vehicleOne);


// old way

function myVehicle(vehicle) {

  const message = 'My ' + vehicle.type + ' is a ' + vehicle.color + ' ' 
+ vehicle.brand + ' ' + vehicle.model + '.';

}
```

Here is the new way of using an object inside a function:

Example

With destructuring:

```
const vehicleOne = {

  brand: 'Ford',

  model: 'Mustang',

  type: 'car',

  year: 2021,

  color: 'red'

}
```

```
myVehicle(vehicleOne);

function myVehicle({type, color, brand, model}) {

  const message = 'My ' + type + ' is a ' + color + ' ' + brand + ' ' +
model + '.';

}
```

```
<!DOCTYPE html>

<html>

<body>

<p id="demo"></p>

<script>

const vehicleOne = {

  brand: 'Ford',

  model: 'Mustang',

  type: 'car',

  year: 2021,

  color: 'red'

}

myVehicle(vehicleOne);

function myVehicle({type, color, brand, model}) {

  const message = 'My ' + type + ' is a ' + color + ' ' + brand + ' ' + model +
'.';

  document.getElementById("demo").innerHTML = message;

}

</script>

</body>
```

</html>

Notice that the object properties do not have to be declared in a specific order.

We can even destructure deeply nested objects by referencing the nested object then using a colon and curly braces to again destructure the items needed from the nested object:

Example

```
const vehicleOne = {

  brand: 'Ford',

  model: 'Mustang',

  type: 'car',

  year: 2021,

  color: 'red',

  registration: {

    city: 'Houston',

    state: 'Texas',

    country: 'USA'

  }

}



myVehicle(vehicleOne)



function myVehicle({ model, registration: { state } }) {

  const message = 'My ' + model + ' is registered in ' + state + '.';

}
```

```
<!DOCTYPE html>
<html>
<body>
<p id="demo"></p>
  <script>
```

```
const vehicleOne = {
  brand: 'Ford',
  model: 'Mustang',
  type: 'car',
  year: 2021,
  color: 'red',
  registration: {
    city: 'Houston',
    state: 'Texas',
    country: 'USA'
  }
}
myVehicle(vehicleOne)
function myVehicle({ model, registration: { state } }) {
  const message = 'My ' + model + ' is registered in ' + state + '.';
  document.getElementById("demo").innerHTML = message;
}
</script>
</body>
</html>
```

Spread Operator

The JavaScript spread operator (`...`) allows us to quickly copy all or part of an existing array or object into another array or object.

```
<!DOCTYPE html>
<html>
<body>
<script>
const numbersOne = [1, 2, 3];
const numbersTwo = [4, 5, 6];
const numbersCombined = [...numbersOne, ...numbersTwo];
document.write(numbersCombined);
</script>
</body>
</html>
```

The spread operator is often used in combination with destructuring.

```
<!DOCTYPE html>
<html>
<body>
<script>
const numbers = [1, 2, 3, 4, 5, 6];
const [one, two, ...rest] = numbers;
document.write("<p>" + one + "</p>");
document.write("<p>" + two + "</p>");
document.write("<p>" + rest + "</p>");
</script>
</body>
</html>
```

Spread Operator

The JavaScript spread operator (`...`) allows us to quickly copy all or part of an existing array or object into another array or object.

Example

```
const numbersOne = [1, 2, 3];

const numbersTwo = [4, 5, 6];

const numbersCombined = [...numbersOne, ...numbersTwo];
```

The spread operator is often used in combination with destructuring.

Example

Assign the first and second items from `numbers` to variables and put the rest in an array:

```
const numbers = [1, 2, 3, 4, 5, 6];

const [one, two, ...rest] = numbers;
```

We can use the spread operator with objects too:

Example

Combine these two objects:

```
const myVehicle = {

  brand: 'Ford',

  model: 'Mustang',

  color: 'red'

}

const updateMyVehicle = {

  type: 'car',
  year: 2021,
  color: 'yellow'
}
const myUpdatedVehicle = {...myVehicle, ...updateMyVehicle}
```

Notice the properties that did not match were combined, but the property that did match, `color`, was overwritten by the last object that was passed, `updateMyVehicle`. The resulting color is now yellow.

React ES6 Modules

Modules

JavaScript modules allow you to break up your code into separate files.

This makes it easier to maintain the code-base.

ES Modules rely on the `import` and `export` statements.

Export

You can export a function or variable from any file.

Let us create a file named `person.js`, and fill it with the things we want to export.

There are two types of exports: Named and Default.

Named Exports

You can create named exports two ways. In-line individually, or all at once at the bottom.

Example

In-line individually:

person.js

```
export const name = "Jesse"

export const age = 40
```

All at once at the bottom:

person.js

```
const name = "Jesse"

const age = 40



export { name, age }
```

Default Exports

Let us create another file, named `message.js`, and use it for demonstrating default export.

You can only have one default export in a file.

Example

message.js

```
const message = () => {

  const name = "Jesse";

  const age = 40;

  return name + ' is ' + age + 'years old.';

};



export default message;
```

Import

You can import modules into a file in two ways, based on if they are named exports or default exports.

Named exports must be destructured using curly braces. Default exports do not.

Example

Import named exports from the file person.js:

```
import { name, age } from "./person.js";
```

Example

Import a default export from the file message.js:

```
import message from "./message.js";
```

Ternary Operator

The ternary operator is a simplified conditional operator like `if` / `else`.

Syntax: `condition ? <expression if true> : <expression if false>`

Here is an example using `if` / `else`:

Example

Before:

```
if (authenticated) {

  renderApp();

} else {

  renderLogin();

}
```

```
<!DOCTYPE html>

<html>

<body>

<h1 id="demo"></h1>

<script>

function renderApp() {

  document.getElementById("demo").innerHTML = "Welcome!";

}

function renderLogin() {

  document.getElementById("demo").innerHTML = "Please log in";

}

let authenticated = true;

if (authenticated) {

  renderApp();

} else {

  renderLogin();

}

</script>

<p>Try changing the "authenticated" variable to false, and run the code to
see what happens.</p>

</body>

</html>
```

Here is the same example using a ternary operator:

Example

With Ternary

```
authenticated ? renderApp() : renderLogin();
```

```
<!DOCTYPE html>
<html>
<body>
<h1 id="demo"></h1>
<script>
function renderApp() {
  document.getElementById("demo").innerHTML = "Welcome!";
}

function renderLogin() {
  document.getElementById("demo").innerHTML = "Please log in";
}

let authenticated = true;

authenticated ? renderApp() : renderLogin();

</script>

<p>Try changing the "authenticated" variable to false, and run the code to see what
happens.</p>
</body>
</html>
```

React JSX

What is JSX?

JSX stands for JavaScript XML.

JSX allows us to write HTML in React.

JSX makes it easier to write and add HTML in React.

Coding JSX

JSX allows us to write HTML elements in JavaScript and place them in the DOM without any `createElement()` and/or `appendChild()` methods.

JSX converts HTML tags into react elements.

You are not required to use JSX, but JSX makes it easier to write React applications.

Here are two examples. The first uses JSX and the second does not:

Example 1

JSX:

```
const myElement = <h1>I Love JSX!</h1>;



const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(myElement);
```

Example 2

Without JSX:

```
const myElement = React.createElement('h1', {}, 'I do not use JSX!');



const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(myElement);
```

As you can see in the first example, JSX allows us to write HTML directly within the JavaScript code.

JSX is an extension of the JavaScript language based on ES6, and is translated into regular JavaScript at runtime.

Expressions in JSX

With JSX you can write expressions inside curly braces `{ }`.

The expression can be a React variable, or property, or any other valid JavaScript expression. JSX will execute the expression and return the result:

Example

Execute the expression `5 + 5`:

```
const myElement = <h1>React is {5 + 5} times better with JSX</h1>;
```

Inserting a Large Block of HTML

To write HTML on multiple lines, put the HTML inside parentheses:

Example

Create a list with three list items:

```
const myElement = (

  <ul>

    <li>Apples</li>

    <li>Bananas</li>

    <li>Cherries</li>

  </ul>

);
```

One Top Level Element

The HTML code must be wrapped in *ONE* top level element.

So if you like to write *two* paragraphs, you must put them inside a parent element, like a `div` element.

Example

Wrap two paragraphs inside one DIV element:

```
const myElement = (

  <div>

    <p>I am a paragraph.</p>

    <p>I am a paragraph too.</p>

  </div>

);
```

JSX will throw an error if the HTML is not correct, or if the HTML misses a parent element.

Alternatively, you can use a "fragment" to wrap multiple lines. This will prevent unnecessarily adding extra nodes to the DOM.

A fragment looks like an empty HTML tag: `<></>`.

Example

Wrap two paragraphs inside a fragment:

```
const myElement = (

  <>

    <p>I am a paragraph.</p>

    <p>I am a paragraph too.</p>

  </>

);
```

Elements Must be Closed

JSX follows XML rules, and therefore HTML elements must be properly closed.

Example

Close empty elements with `/>`

```
const myElement = <input type="text" />;
```

JSX will throw an error if the HTML is not properly closed.

Attribute class = className

The `class` attribute is a much used attribute in HTML, but since JSX is rendered as JavaScript, and the `class` keyword is a reserved word in JavaScript, you are not allowed to use it in JSX.

Use attribute `className` instead.

JSX solved this by using `className` instead. When JSX is rendered, it translates `className` attributes into `class` attributes.

Example

Use attribute `className` instead of `class` in JSX:

```
const myElement = <h1 className="myclass">Hello World</h1>;
```

Conditions - if statements

React supports `if` statements, but not *inside* JSX.

To be able to use conditional statements in JSX, you should put the `if` statements outside of the JSX, or you could use a ternary expression instead:

Option 1:

Write `if` statements outside of the JSX code:

Example

Write "Hello" if `x` is less than 10, otherwise "Goodbye":

```
const x = 5;

let text = "Goodbye";

if (x < 10) {

  text = "Hello";
}


const myElement = <h1>{text}</h1>;
```

Option 2:

Use ternary expressions instead:

Example

Write "Hello" if `x` is less than 10, otherwise "Goodbye":

```
const x = 5;

const myElement = <h1>{(x) < 10 ? "Hello" : "Goodbye"}</h1>;
```

Note that in order to embed a JavaScript expression inside JSX, the JavaScript must be wrapped with curly braces, `{}`.

React Class Components

Before React 16.8, Class components were the only way to track state and lifecycle on a React component. Function components were considered "state-less".

With the addition of Hooks, Function components are now almost equivalent to Class components. The differences are so minor that you will probably never need to use a Class component in React.

Even though Function components are preferred, there are no current plans on removing Class components from React.

This section will give you an overview of how to use Class components in React.

Feel free to skip this section, and use Function Components instead.

React Components

Components are independent and reusable bits of code. They serve the same purpose as JavaScript functions, but work in isolation and return HTML via a render() function.

Components come in two types, Class components and Function components, in this chapter you will learn about Class components.

Create a Class Component

When creating a React component, the component's name must start with an upper case letter.

The component has to include the `extends React.Component` statement, this statement creates an inheritance to React.Component, and gives your component access to React.Component's functions.

The component also requires a `render()` method, this method returns HTML.

Example

Create a Class component called `Car`

```
class Car extends React.Component {
  render() {
    return <h2>Hi, I am a Car!</h2>;
  }
}
```

Now your React application has a component called Car, which returns a `<h2>` element.

To use this component in your application, use similar syntax as normal HTML: `<Car />`

Example

Display the `Car` component in the "root" element:

```
const root = ReactDOM.createRoot(document.getElementById('root'));
```

```
root.render(<Car />)
```

Component Constructor

If there is a `constructor()` function in your component, this function will be called when the component gets initiated.

The constructor function is where you initiate the component's properties.

In React, component properties should be kept in an object called `state`.

You will learn more about `state` later in this tutorial.

The constructor function is also where you honor the inheritance of the parent component by including the `super()` statement, which executes the parent component's constructor function, and your component has access to all the functions of the parent component (`React.Component`).

Example

Create a constructor function in the Car component, and add a color property:

```
class Car extends React.Component {

  constructor() {

    super();

    this.state = {color: "red"};

  }

  render() {

    return <h2>I am a Car!</h2>;

  }

}
```

Use the color property in the render() function:

Example

```
class Car extends React.Component {
  constructor() {
    super();
    this.state = {color: "red"};
  }
  render() {
    return <h2>I am a {this.state.color} Car!</h2>;
  }
}
```

Props

Another way of handling component properties is by using `props`.

Props are like function arguments, and you send them into the component as attributes.

You will learn more about `props` in the next chapter.

Example

Use an attribute to pass a color to the Car component, and use it in the render() function:

```
class Car extends React.Component {
  render() {
    return <h2>I am a {this.props.color} Car!</h2>;
  }
}


const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car color="red"/>)
```

Props in the Constructor

If your component has a constructor function, the props should always be passed to the constructor and also to the React.Component via the `super()` method.

Example

```
class Car extends React.Component {

  constructor(props) {

    super(props);

  }

  render() {

    return <h2>I am a {this.props.model}!</h2>;

  }

}

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Car model="Mustang"/>);
```

Components in Components

We can refer to components inside other components:

Example

Use the Car component inside the Garage component:

```
class Car extends React.Component {

  render() {

    return <h2>I am a Car!</h2>;

  }

}
```

```
class Garage extends React.Component {
  render() {
    return (
      <div>
      <h1>Who lives in my Garage?</h1>
      <Car />
      </div>
    );
  }
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Garage />);
```

Components in Files

React is all about re-using code, and it can be smart to insert some of your components in separate files.

To do that, create a new file with a `.js` file extension and put the code inside it:

Note that the file must start by importing React (as before), and it has to end with the statement `export default Car;`.

Example

This is the new file, we named it `Car.js`:

```
import React from 'react';


class Car extends React.Component {
  render() {
    return <h2>Hi, I am a Car!</h2>;
  }
}


export default Car;
```

To be able to use the `Car` component, you have to import the file in your application.

Example

Now we import the `Car.js` file in the application, and we can use the `Car` component as if it was created here.

```
import React from 'react';

import ReactDOM from 'react-dom/client';

import Car from './Car.js';



const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Car />);
```

React Class Component State

React Class components have a built-in `state` object.

You might have noticed that we used `state` earlier in the component constructor section.

The `state` object is where you store property values that belongs to the component.

When the `state` object changes, the component re-renders.

Creating the state Object

The state object is initialized in the constructor:

Example

Specify the `state` object in the constructor method:

```
class Car extends React.Component {
  constructor(props) {
    super(props);
  this.state = {brand: "Ford"};
  }
  render() {
    return (
      <div>
        <h1>My Car</h1>
      </div>
    );
  }
}
```

The state object can contain as many properties as you like:

Example

Specify all the properties your component need:

```
class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Mustang",
      color: "red",
      year: 1964
    };
  }
  render() {
    return (
      <div>
        <h1>My Car</h1>
      </div>
    );
  }
}
```

Using the `state` Object

Refer to the `state` object anywhere in the component by using the `this.state.`*`propertyname`* syntax:

Example:

Refer to the `state` object in the `render()` method:

```
class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Mustang",
      color: "red",
      year: 1964
    };
  }
  render() {
    return (
      <div>
        <h1>My {this.state.brand}</h1>
        <p>
          It is a {this.state.color}
          {this.state.model}
          from {this.state.year}.
        </p>
      </div>
    );
  }
}
```

Changing the `state` Object

To change a value in the state object, use the `this.setState()` method.

When a value in the `state` object changes, the component will re-render, meaning that the output will change according to the new value(s).

Example:

Add a button with an `onClick` event that will change the color property:

```
class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Mustang",
      color: "red",
      year: 1964
    };
  }
  changeColor = () => {
    this.setState({color: "blue"});
  }
  render() {
    return (
      <div>
        <h1>My {this.state.brand}</h1>
        <p>
          It is a {this.state.color}
          {this.state.model}
          from {this.state.year}.
        </p>
        <button
          type="button"
          onClick={this.changeColor}
        >Change color</button>
      </div>
    );
  }
}
```

Always use the `setState()` method to change the state object, it will ensure that the component knows its been updated and calls the render() method (and all the other lifecycle methods).

Lifecycle of Components

Each component in React has a lifecycle which you can monitor and manipulate during its three main phases.

The three phases are: **Mounting**, **Updating**, and **Unmounting**.

Mounting

Mounting means putting elements into the DOM.

React has four built-in methods that gets called, in this order, when mounting a component:

1. `constructor()`
2. `getDerivedStateFromProps()`
3. `render()`
4. `componentDidMount()`

The `render()` method is required and will always be called, the others are optional and will be called if you define them.

constructor

The `constructor()` method is called before anything else, when the component is initiated, and it is the natural place to set up the initial `state` and other initial values.

The `constructor()` method is called with the `props`, as arguments, and you should always start by calling the `super(props)` before anything else, this will initiate the parent's constructor method and allows the component to inherit methods from its parent (`React.Component`).

Example:

The `constructor` method is called, by React, every time you make a component:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header />);
```

getDerivedStateFromProps

The `getDerivedStateFromProps()` method is called right before rendering the element(s) in the DOM.

This is the natural place to set the `state` object based on the initial `props`.

It takes `state` as an argument, and returns an object with changes to the `state`.

The example below starts with the favorite color being "red", but the `getDerivedStateFromProps()` method updates the favorite color based on the `favcol` attribute:

Example:

The `getDerivedStateFromProps` method is called right before the render method:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  static getDerivedStateFromProps(props, state) {
    return {favoritecolor: props.favcol };
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header favcol="yellow"/>);
```

render

The `render()` method is required, and is the method that actually outputs the HTML to the DOM.

Example:

A simple component with a simple `render()` method:

```
class Header extends React.Component {
  render() {
    return (
      <h1>This is the content of the Header component</h1>
    );
  }
}


const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Header />);
```

componentDidMount

The `componentDidMount()` method is called after the component is rendered.

This is where you run statements that requires that the component is already placed in the DOM.

Example:

At first my favorite color is red, but give me a second, and it is yellow instead:

```
class Header extends React.Component {

  constructor(props) {

    super(props);

    this.state = {favoritecolor: "red"};

  }

  componentDidMount() {

    setTimeout(() => {

      this.setState({favoritecolor: "yellow"})

    }, 1000)

  }

  render() {

    return (

      <h1>My Favorite Color is {this.state.favoritecolor}</h1>

    );

  }
}


const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header />);
```

Updating

The next phase in the lifecycle is when a component is *updated*.

A component is updated whenever there is a change in the component's `state` or `props`.

React has five built-in methods that gets called, in this order, when a component is updated:

1. `getDerivedStateFromProps()`
2. `shouldComponentUpdate()`
3. `render()`
4. `getSnapshotBeforeUpdate()`
5. `componentDidUpdate()`

The `render()` method is required and will always be called, the others are optional and will be called if you define them.

getDerivedStateFromProps

Also at *updates* the `getDerivedStateFromProps` method is called. This is the first method that is called when a component gets updated.

This is still the natural place to set the `state` object based on the initial props.

The example below has a button that changes the favorite color to blue, but since the `getDerivedStateFromProps()` method is called, which updates the state with the color from the favcol attribute, the favorite color is still rendered as yellow:

Example:

If the component gets updated, the `getDerivedStateFromProps()` method is called:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  static getDerivedStateFromProps(props, state) {
    return {favoritecolor: props.favcol };
  }
  changeColor = () => {
    this.setState({favoritecolor: "blue"});
  }
  render() {
    return (
      <div>
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
      <button type="button" onClick={this.changeColor}>Change
color</button>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header favcol="yellow" />);
```

shouldComponentUpdate

In the `shouldComponentUpdate()` method you can return a Boolean value that specifies whether React should continue with the rendering or not.

The default value is `true`.

The example below shows what happens when the `shouldComponentUpdate()` method returns `false`:

Example:

Stop the component from rendering at any update:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  shouldComponentUpdate() {
    return false;
  }
  changeColor = () => {
    this.setState({favoritecolor: "blue"});
  }
  render() {
    return (
      <div>
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
      <button type="button" onClick={this.changeColor}>Change color</button>
      </div>
    );
  }
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header />);
```

Example:

Same example as above, but this time the `shouldComponentUpdate()` method returns `true` instead:

```
class Header extends React.Component {

  constructor(props) {

    super(props);

    this.state = {favoritecolor: "red"};

  }

  shouldComponentUpdate() {

    return true;

  }

  changeColor = () => {

    this.setState({favoritecolor: "blue"});

  }

  render() {

    return (

      <div>

      <h1>My Favorite Color is {this.state.favoritecolor}</h1>

      <button type="button" onClick={this.changeColor}>Change
color</button>

      </div>

    );

  }

}



const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Header />);
```

render

The `render()` method is of course called when a component gets *updated*, it has to re-render the HTML to the DOM, with the new changes.

The example below has a button that changes the favorite color to blue:

Example:

Click the button to make a change in the component's state:

```
class Header extends React.Component {

  constructor(props) {

    super(props);

    this.state = {favoritecolor: "red"};

  }

  changeColor = () => {

    this.setState({favoritecolor: "blue"});

  }

  render() {

    return (

      <div>

      <h1>My Favorite Color is {this.state.favoritecolor}</h1>

      <button type="button" onClick={this.changeColor}>Change
color</button>

      </div>

    );

  }

}



const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Header />);
```

getSnapshotBeforeUpdate

In the `getSnapshotBeforeUpdate()` method you have access to the `props` and `state` *before* the update, meaning that even after the update, you can check what the values were *before* the update.

If the `getSnapshotBeforeUpdate()` method is present, you should also include the `componentDidUpdate()` method, otherwise you will get an error.

The example below might seem complicated, but all it does is this:

When the component is *mounting* it is rendered with the favorite color "red".

When the component *has been mounted,* a timer changes the state, and after one second, the favorite color becomes "yellow".

This action triggers the *update* phase, and since this component has a `getSnapshotBeforeUpdate()` method, this method is executed, and writes a message to the empty DIV1 element.

Then the `componentDidUpdate()` method is executed and writes a message in the empty DIV2 element:

Example:

Use the `getSnapshotBeforeUpdate()` method to find out what the `state` object looked like before the update:

```
class Header extends React.Component {

  constructor(props) {

    super(props);

    this.state = {favoritecolor: "red"};

  }

  componentDidMount() {

    setTimeout(() => {

      this.setState({favoritecolor: "yellow"})

    }, 1000)
  }
  getSnapshotBeforeUpdate(prevProps, prevState) {
    document.getElementById("div1").innerHTML =
    "Before the update, the favorite was " + prevState.favoritecolor;
  }
  componentDidUpdate() {
    document.getElementById("div2").innerHTML =
    "The updated favorite is " + this.state.favoritecolor;
  }
  render() {
    return (
      <div>
        <h1>My Favorite Color is {this.state.favoritecolor}</h1>
        <div id="div1"></div>
        <div id="div2"></div>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header />);
```

componentDidUpdate

The `componentDidUpdate` method is called after the component is updated in the DOM.

The example below might seem complicated, but all it does is this:

When the component is *mounting* it is rendered with the favorite color "red".

When the component *has been mounted,* a timer changes the state, and the color becomes "yellow".

This action triggers the *update* phase, and since this component has a `componentDidUpdate` method, this method is executed and writes a message in the empty DIV element:

Example:

The `componentDidUpdate` method is called after the update has been rendered in the DOM:

```
class Header extends React.Component {

  constructor(props) {

    super(props);

    this.state = {favoritecolor: "red"};

  }

  componentDidMount() {

    setTimeout(() => {

      this.setState({favoritecolor: "yellow"})

    }, 1000)

  }

  componentDidUpdate() {

    document.getElementById("mydiv").innerHTML =

    "The updated favorite is " + this.state.favoritecolor;

  }

  render() {

    return (

      <div>

      <h1>My Favorite Color is {this.state.favoritecolor}</h1>

      <div id="mydiv"></div>
      </div>
    );
  }
}


const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header />);
```

Unmounting

The next phase in the lifecycle is when a component is removed from the DOM, or *unmounting* as React likes to call it.

React has only one built-in method that gets called when a component is unmounted:

- `componentWillUnmount()`

componentWillUnmount

The `componentWillUnmount` method is called when the component is about to be removed from the DOM.

Example:

Click the button to delete the header:

```
class Container extends React.Component {
  constructor(props) {
    super(props);
    this.state = {show: true};
  }
  delHeader = () => {
    this.setState({show: false});
  }
  render() {
    let myheader;
    if (this.state.show) {
      myheader = <Child />;
    };
    return (
      <div>
      {myheader}
      <button type="button" onClick={this.delHeader}>Delete
Header</button>
      </div>
    );
  }
}

class Child extends React.Component {
  componentWillUnmount() {
    alert("The component named Header is about to be unmounted.");
  }
  render() {
    return (
      <h1>Hello World!</h1>
    );
  }
}
const root = ReactDOM.createRoot(document.getElementById('root'));
```

```
root.render(<Container />);
```

React Events

Just like HTML DOM events, React can perform actions based on user events.

React has the same events as HTML: click, change, mouseover etc.

Adding Events

React events are written in camelCase syntax:

`onClick` instead of `onclick`.

React event handlers are written inside curly braces:

`onClick={shoot}` instead of `onclick="shoot()"`.

React:

```
<button onClick={shoot}>Take the Shot!</button>
```

HTML:

```
<button onclick="shoot()">Take the Shot!</button>
```

Example:

Put the `shoot` function inside the `Football` component:

```
function Football() {
  const shoot = () => {
    alert("Great Shot!");
  }
  return (
    <button onClick={shoot}>Take the shot!</button>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Football />);
```

Passing Arguments

To pass an argument to an event handler, use an arrow function.

Example:

Send "Goal!" as a parameter to the shoot function, using arrow function:

```
function Football() {
  const shoot = (a) => {
    alert(a);
  }

  return (
    <button onClick={() => shoot("Goal!")}>Take the shot!</button>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Football />);
```

```
import React from 'react';
import ReactDOM from 'react-dom/client';

function Football() {
  const shoot = (a) => {
    alert(a);
  }

  return (
    <button onClick={() => shoot("Goal!")}>Take the shot!</button>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Football />);
```

React Event Object

Event handlers have access to the React event that triggered the function.

In our example the event is the "click" event.

Example:

Arrow Function: Sending the event object manually:

```
function Football() {

  const shoot = (a, b) => {

    alert(b.type);

    /*

    'b' represents the React event that triggered the function,

    in this case the 'click' event

    */

  }


  return (
    <button onClick={(event) => shoot("Goal!", event)}>Take the
shot!</button>

  );

}



const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Football />);
```

```
import React from 'react';
import ReactDOM from 'react-dom/client';

function Football() {
  const shoot = (a, b) => {
    alert(b.type);
              /*
              'b' represents the React event that triggered the
function.
    In this case, the 'click' event
              */
  }

  return (
    <button onClick={(event) => shoot("Goal!", event)}>Take the
shot!</button>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Football />);
```

React Conditional Rendering

In React, you can conditionally render components.

There are several ways to do this.

`if` Statement

We can use the `if` JavaScript operator to decide which component to render.

Example:

We'll use these two components:

```
function MissedGoal() {

  return <h1>MISSED!</h1>;

}



function MadeGoal() {

  return <h1>Goal!</h1>;

}
```

Example:

Now, we'll create another component that chooses which component to render based on a condition:

```
function Goal(props) {

  const isGoal = props.isGoal;

  if (isGoal) {

    return <MadeGoal/>;

  }

  return <MissedGoal/>;

}



const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Goal isGoal={false} />);
```

```
import React from 'react';
import ReactDOM from 'react-dom/client';

function MissedGoal() {
	return <h1>MISSED!</h1>;
}

function MadeGoal() {
	return <h1>GOAL!</h1>;
}

function Goal(props) {
  const isGoal = props.isGoal;
  if (isGoal) {
    return <MadeGoal/>;
  }
  return <MissedGoal/>;
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Goal isGoal={false} />);
```

```
import React from 'react';
import ReactDOM from 'react-dom/client';

function MissedGoal() {
	return <h1>MISSED!</h1>;
}

function MadeGoal() {
	return <h1>GOAL!</h1>;
}

function Goal(props) {
  const isGoal = props.isGoal;
  if (isGoal) {
    return <MadeGoal/>;
  }
  return <MissedGoal/>;
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Goal isGoal={false} />);
```

Try changing the `isGoal` attribute to `true`:

Example:

```
const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Goal isGoal={true} />);

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

function MissedGoal() {
	return <h1>MISSED!</h1>;
}

function MadeGoal() {
	return <h1>GOAL!</h1>;
}

function Goal(props) {
  const isGoal = props.isGoal;
  if (isGoal) {
    return <MadeGoal/>;
  }
  return <MissedGoal/>;
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Goal isGoal={true} />);
```

Logical && Operator

Another way to conditionally render a React component is by using the `&&` operator.

Example:

We can embed JavaScript expressions in JSX by using curly braces:

```
function Garage(props) {

  const cars = props.cars;

  return (

    <>

      <h1>Garage</h1>
      {cars.length > 0 &&

        <h2>

          You have {cars.length} cars in your garage.

        </h2>

      }

    </>

  );

}



const cars = ['Ford', 'BMW', 'Audi'];

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Garage cars={cars} />);
```

```
import React from 'react';
import ReactDOM from 'react-dom/client';

function Garage(props) {
  const cars = props.cars;
  return (
    <>
      <h1>Garage</h1>
      {cars.length > 0 &&
        <h2>
          You have {cars.length} cars in your garage.
        </h2>
      }
    </>
  );
}

const cars = ['Ford', 'BMW', 'Audi'];
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Garage cars={cars} />);
```

If `cars.length > 0` is equates to true, the expression after `&&` will render.

Try emptying the `cars` array:

Example:

```
const cars = [];

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Garage cars={cars} />);
```

```
import React from 'react';
import ReactDOM from 'react-dom/client';

function Garage(props) {
  const cars = props.cars;
  return (
    <>
      <h1>Garage</h1>
      {cars.length > 0 &&
        <h2>
          You have {cars.length} cars in your garage.
        </h2>
      }
    </>
  );
}

const cars = [];
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Garage cars={cars} />);
```

Ternary Operator

Another way to conditionally render elements is by using a ternary operator.

```
condition ? true : false
```

We will go back to the goal example.

Example:

Return the `MadeGoal` component if `isGoal` is `true`, otherwise return the `MissedGoal` component:

```
function Goal(props) {

  const isGoal = props.isGoal;

  return (
    <>

      { isGoal ? <MadeGoal/> : <MissedGoal/> }

    </>

  );

}



const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Goal isGoal={false} />);
```

```
import React from 'react';
import ReactDOM from 'react-dom/client';

function MissedGoal() {
	return <h1>MISSED!</h1>;
}

function MadeGoal() {
	return <h1>GOAL!</h1>;
}

function Goal(props) {
  const isGoal = props.isGoal;
	return (
		<>
			{ isGoal ? <MadeGoal/> : <MissedGoal/> }
		</>
	);
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Goal isGoal={false} />);
```

React Lists

In React, you will render lists with some type of loop.

The JavaScript `map()` array method is generally the preferred method.

If you need a refresher on the `map()` method, check out the ES6 section.

Example:

Let's render all of the cars from our garage:

```
function Car(props) {

  return <li>I am a { props.brand }</li>;

}



function Garage() {

  const cars = ['Ford', 'BMW', 'Audi'];

  return (

    <>

      <h1>Who lives in my garage?</h1>

      <ul>

        {cars.map((car) => <Car brand={car} />)}

      </ul>

    </>

  );

}



const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Garage />);
```

```
import React from 'react';
import ReactDOM from 'react-dom/client';

function Car(props) {
  return <li>I am a { props.brand }</li>;
}

function Garage() {
  const cars = ['Ford', 'BMW', 'Audi'];
  return (
    <>
          <h1>Who lives in my garage?</h1>
          <ul>
```

```
        {cars.map((car) => <Car brand={car} />)}
      </ul>
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Garage />);

/*
If you run this example in your create-react-app,
you will receive a warning that there is no "key" provided for the list
items.
*/
```

When you run this code in your `create-react-app`, it will work but you will receive a warning that there is no "key" provided for the list items

Keys

Keys allow React to keep track of elements. This way, if an item is updated or removed, only that item will be re-rendered instead of the entire list.

Keys need to be unique to each sibling. But they can be duplicated globally.

Generally, the key should be a unique ID assigned to each item. As a last resort, you can use the array index as a key.

Example:

Let's refactor our previous example to include keys:

```
function Car(props) {

  return <li>I am a { props.brand }</li>;

}



function Garage() {

  const cars = [

    {id: 1, brand: 'Ford'},

    {id: 2, brand: 'BMW'},
    {id: 3, brand: 'Audi'}

  ];

  return (

    <>

      <h1>Who lives in my garage?</h1>

      <ul>

        {cars.map((car) => <Car key={car.id} brand={car.brand} />)}

      </ul>

    </>

  );

}



const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Garage />);
```

```
import React from 'react';
import ReactDOM from 'react-dom/client';

function Car(props) {
  return <li>I am a { props.brand }</li>;
}

function Garage() {
  const cars = [
    {id: 1, brand: 'Ford'},
    {id: 2, brand: 'BMW'},
    {id: 3, brand: 'Audi'}
  ];
  return (
    <>
           <h1>Who lives in my garage?</h1>
           <ul>
        {cars.map((car) => <Car key={car.id} brand={car.brand} />)}
      </ul>
    </>
  );
}
```

React Forms

Just like in HTML, React uses forms to allow users to interact with the web page.

Adding Forms in React

You add a form with React like any other element:

Example:

Add a form that allows users to enter their name:

```
function MyForm() {

  return (

    <form>

      <label>Enter your name:

        <input type="text" />

      </label>

    </form>

  )

}

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<MyForm />);
```

```
import React from 'react';
import ReactDOM from 'react-dom/client';

function MyForm() {
  return (
    <form>
      <label>Enter your name:
        <input type="text" />
      </label>
    </form>
  )
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<MyForm />);
```

This will work as normal, the form will submit and the page will refresh.

But this is generally not what we want to happen in React.

We want to prevent this default behavior and let React control the form.

Handling Forms

Handling forms is about how you handle the data when it changes value or gets submitted.

In HTML, form data is usually handled by the DOM.

In React, form data is usually handled by the components.

When the data is handled by the components, all the data is stored in the component state.

You can control changes by adding event handlers in the `onChange` attribute.

We can use the `useState` Hook to keep track of each inputs value and provide a "single source of truth" for the entire application.

See the React Hooks section for more information on Hooks.

Example:

Use the `useState` Hook to manage the input:

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

import ReactDOM from 'react-dom/client';



function MyForm() {

  const [name, setName] = useState("");
  return (

    <form>

      <label>Enter your name:

        <input

          type="text"

          value={name}

          onChange={(e) => setName(e.target.value)}

        />

      </label>

    </form>

  )

}



const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<MyForm />);
```

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

function MyForm() {
  const [name, setName] = useState("");

  return (
    <form>
      <label>Enter your name:
        <input
          type="text"
          value={name}
          onChange={(e) => setName(e.target.value)}
        />
      </label>
    </form>
  )
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<MyForm />);
```

Submitting Forms

You can control the submit action by adding an event handler in the `onSubmit` attribute for the `<form>`:

Example:

Add a submit button and an event handler in the `onSubmit` attribute:

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

import ReactDOM from 'react-dom/client';


function MyForm() {

  const [name, setName] = useState("");


  const handleSubmit = (event) => {

    event.preventDefault();

    alert(`The name you entered was: ${name}`)

  }


  return (

    <form onSubmit={handleSubmit}>

      <label>Enter your name:

        <input

          type="text"

          value={name}

          onChange={(e) => setName(e.target.value)}

        />

      </label>

      <input type="submit" />
    </form>

  )

}



const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<MyForm />);
```

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

function MyForm() {
  const [name, setName] = useState("");

  const handleSubmit = (event) => {
    event.preventDefault();
    alert(`The name you entered was: ${name}`);
  }

  return (
    <form onSubmit={handleSubmit}>
      <label>Enter your name:
        <input
          type="text"
          value={name}
          onChange={(e) => setName(e.target.value)}
        />
      </label>
      <input type="submit" />
    </form>
  )
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<MyForm />);
```

Multiple Input Fields

You can control the values of more than one input field by adding a `name` attribute to each element.

We will initialize our state with an empty object.

To access the fields in the event handler use the `event.target.name` and `event.target.value` syntax.

To update the state, use square brackets [bracket notation] around the property name.

Example:

Write a form with two input fields:

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

import ReactDOM from 'react-dom/client';


function MyForm() {

  const [inputs, setInputs] = useState({});


  const handleChange = (event) => {

    const name = event.target.name;

    const value = event.target.value;

    setInputs(values => ({...values, [name]: value}))

  }


  const handleSubmit = (event) => {

    event.preventDefault();

    alert(inputs);

  }


  return (

    <form onSubmit={handleSubmit}>

      <label>Enter your name:

      <input

        type="text"
```

```
          name="username"

          value={inputs.username || ""}

          onChange={handleChange}

        />

        </label>

        <label>Enter your age:

          <input

            type="number"

            name="age"

            value={inputs.age || ""}

            onChange={handleChange}

          />

          </label>

          <input type="submit" />

      </form>

  )

}


const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<MyForm />);
```

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

function MyForm() {
  const [inputs, setInputs] = useState({});

  const handleChange = (event) => {
    const name = event.target.name;
    const value = event.target.value;
    setInputs(values => ({...values, [name]: value}))
  }

  const handleSubmit = (event) => {
    event.preventDefault();
```

```
      console.log(inputs);
    }

    return (
      <form onSubmit={handleSubmit}>
        <label>Enter your name:
        <input
          type="text"
          name="username"
          value={inputs.username || ""}
          onChange={handleChange}
        />
        </label>
        <label>Enter your age:
          <input
            type="number"
            name="age"
            value={inputs.age || ""}
            onChange={handleChange}
          />
          </label>
          <input type="submit" />
      </form>
    )
  }

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<MyForm />);

/*
Click F12 and navigate to the "Console view"
to see the result when you submit the form.
*/
```

Note: We use the same event handler function for both input fields, we could write one event handler for each, but this gives us much cleaner code and is the preferred way in React.

Textarea

The textarea element in React is slightly different from ordinary HTML.

In HTML the value of a textarea was the text between the start tag `<textarea>` and the end tag `</textarea>`.

```
<textarea>
```

```
  Content of the textarea.
</textarea>
```

In React the value of a textarea is placed in a value attribute. We'll use the `useState` Hook to manage the value of the textarea:

Example:

A simple textarea with some content:

```
import { useState } from 'react';
import ReactDOM from 'react-dom/client';

function MyForm() {
  const [textarea, setTextarea] = useState(
    "The content of a textarea goes in the value attribute"
  );

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

  return (
    <form>
      <textarea value={textarea} onChange={handleChange} />
    </form>
  )
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<MyForm />);
```

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

function MyForm() {
  const [textarea, setTextarea] = useState(
    "The content of a textarea goes in the value attribute"
  );

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

  return (
    <form>
      <textarea value={textarea} onChange={handleChange} />
    </form>
  )
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<MyForm />);
```

Select

A drop down list, or a select box, in React is also a bit different from HTML.

In HTML, the selected value in the drop down list was defined with the `selected` attribute:

HTML:

```
<select>

  <option value="Ford">Ford</option>

  <option value="Volvo" selected>Volvo</option>

  <option value="Fiat">Fiat</option>

</select>
```

In React, the selected value is defined with a `value` attribute on the `select` tag:

Example:

A simple select box, where the selected value "Volvo" is initialized in the constructor:

```
function MyForm() {

  const [myCar, setMyCar] = useState("Volvo");



  const handleChange = (event) => {

    setMyCar(event.target.value)

  }



  return (

    <form>

      <select value={myCar} onChange={handleChange}>

        <option value="Ford">Ford</option>

        <option value="Volvo">Volvo</option>

        <option value="Fiat">Fiat</option>

      </select>

    </form>

  )

}
```

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

function MyForm() {
  const [myCar, setMyCar] = useState("Volvo");

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

  return (
    <form>
      <select value={myCar} onChange={handleChange}>
        <option value="Ford">Ford</option>
        <option value="Volvo">Volvo</option>
        <option value="Fiat">Fiat</option>
      </select>
    </form>
  )
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<MyForm />);
```

By making these slight changes to `<textarea>` and `<select>`, React is able to handle all input elements in the same way.

React Router

Create React App doesn't include page routing.

React Router is the most popular solution.

Add React Router

To add React Router in your application, run this in the terminal from the root directory of the application:

```
npm i -D react-router-dom
```

Note: This tutorial uses React Router v6.

If you are upgrading from v5, you will need to use the @latest flag:

```
npm i -D react-router-dom@latest
```

Folder Structure

To create an application with multiple page routes, let's first start with the file structure.

Within the `src` folder, we'll create a folder named `pages` with several files:

`src\pages\`:

- `Layout.js`
- `Home.js`
- `Blogs.js`
- `Contact.js`
- `NoPage.js`

Each file will contain a very basic React component.

Basic Usage

Now we will use our Router in our `index.js` file.

Example

Use React Router to route to pages based on URL:

`index.js`:

```
import ReactDOM from "react-dom/client";

import { BrowserRouter, Routes, Route } from "react-router-dom";

import Layout from "./pages/Layout";

import Home from "./pages/Home";

import Blogs from "./pages/Blogs";

import Contact from "./pages/Contact";

import NoPage from "./pages/NoPage";


export default function App() {

  return (

    <BrowserRouter>

      <Routes>

        <Route path="/" element={<Layout />}>

          <Route index element={<Home />} />

          <Route path="blogs" element={<Blogs />} />
          <Route path="contact" element={<Contact />} />

          <Route path="*" element={<NoPage />} />

        </Route>

      </Routes>

    </BrowserRouter>

  );

}



const root = ReactDOM.createRoot(document.getElementById('root'));

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

```
import ReactDOM from "react-dom/client";
import { BrowserRouter, Routes, Route } from "react-router-dom";
import Layout from "./pages/Layout";
import Home from "./pages/Home";
import Blogs from "./pages/Blogs";
import Contact from "./pages/Contact";
import NoPage from "./pages/NoPage";

export default function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Layout />}>
          <Route index element={<Home />} />
          <Route path="blogs" element={<Blogs />} />
          <Route path="contact" element={<Contact />} />
          <Route path="*" element={<NoPage />} />
        </Route>
      </Routes>
    </BrowserRouter>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

Example Explained

We wrap our content first with `<BrowserRouter>`.

Then we define our `<Routes>`. An application can have multiple `<Routes>`. Our basic example only uses one.

`<Route>`s can be nested. The first `<Route>` has a path of `/` and renders the `Layout` component.

The nested `<Route>`s inherit and add to the parent route. So the `blogs` path is combined with the parent and becomes `/blogs`.

The `Home` component route does not have a path but has an `index` attribute. That specifies this route as the default route for the parent route, which is `/`.

Setting the `path` to `*` will act as a catch-all for any undefined URLs. This is great for a 404 error page.

Pages / Components

The `Layout` component has `<Outlet>` and `<Link>` elements.

The `<Outlet>` renders the current route selected.

`<Link>` is used to set the URL and keep track of browsing history.

Anytime we link to an internal path, we will use `<Link>` instead of `<a href="">`.

The "layout route" is a shared component that inserts common content on all pages, such as a navigation menu.

`Layout.js`:

```
import { Outlet, Link } from "react-router-dom";


const Layout = () => {
  return (
    <>
      <nav>
        <ul>
```

```
          <li>
            <Link to="/">Home</Link>
          </li>
          <li>
            <Link to="/blogs">Blogs</Link>
          </li>
          <li>
            <Link to="/contact">Contact</Link>
          </li>
        </ul>
      </nav>

      <Outlet />
    </>
  )
};

export default Layout;
```

`Home.js`:

```
const Home = () => {
  return <h1>Home</h1>;
};

export default Home;
```

`Blogs.js`:

```
const Blogs = () => {
  return <h1>Blog Articles</h1>;
};


export default Blogs;
```

`Contact.js`:

```
const Contact = () => {
  return <h1>Contact Me</h1>;
};


export default Contact;
```

`NoPage.js`:

```
const NoPage = () => {
  return <h1>404</h1>;
};


export default NoPage;
```

React Memo

Using `memo` will cause React to skip rendering a component if its props have not changed.

This can improve performance.

This section uses React Hooks. See the React Hooks section for more information on Hooks.

Problem

In this example, the `Todos` component re-renders even when the todos have not changed.

Example:

index.js:

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

import ReactDOM from "react-dom/client";

import Todos from "./Todos";


const App = () => {

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

  const [todos, setTodos] = useState(["todo 1", "todo 2"]);


  const increment = () => {

    setCount((c) => c + 1);

  };


  return (

    <>

      <Todos todos={todos} />

      <hr />

      <div>

        Count: {count}

        <button onClick={increment}>+</button>

      </div>

    </>

  );

};


const root = ReactDOM.createRoot(document.getElementById('root'));
```

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

`Todos.js`:

```
const Todos = ({ todos }) => {

  console.log("child render");

  return (

    <>

      <h2>My Todos</h2>

      {todos.map((todo, index) => {

        return <p key={index}>{todo}</p>;

      })}

    </>

  );

};



export default Todos;
```

```
import { useState } from "react";
import ReactDOM from "react-dom/client";
import Todos from "./Todos";

const App = () => {
  const [count, setCount] = useState(0);
  const [todos, setTodos] = useState(["todo 1", "todo 2"]);

  const increment = () => {
    setCount((c) => c + 1);
  };

  return (
    <>
      <Todos todos={todos} />
      <hr />
      <div>
        Count: {count}
        <button onClick={increment}>+</button>
      </div>
    </>
  );
};
```

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

When you click the increment button, the Todos component re-renders.

If this component was complex, it could cause performance issues.

Solution

To fix this, we can use memo.

Use memoto keep the Todos component from needlessly re-rendering.

Wrap the Todos component export in memo:

Example:

index.js:

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

import ReactDOM from "react-dom/client";

import Todos from "./Todos";


const App = () => {

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

  const [todos, setTodos] = useState(["todo 1", "todo 2"]);


  const increment = () => {

    setCount((c) => c + 1);

  };


  return (

    <>
```

```
      <Todos todos={todos} />
      <hr />
      <div>
        Count: {count}
        <button onClick={increment}>+</button>
      </div>
    </>
  );
};

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

Todos.js:

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

const Todos = ({ todos }) => {
  console.log("child render");
  return (
    <>
      <h2>My Todos</h2>
      {todos.map((todo, index) => {
        return <p key={index}>{todo}</p>;
      })}
    </>
  );
};

```

```
export default memo(Todos);

import { useState } from "react";
import ReactDOM from "react-dom/client";
import Todos from "./Todos";

const App = () => {
  const [count, setCount] = useState(0);
  const [todos, setTodos] = useState(["todo 1", "todo 2"]);

  const increment = () => {
    setCount((c) => c + 1);
  };

  return (
    <>
      <Todos todos={todos} />
      <hr />
      <div>
        Count: {count}
        <button onClick={increment}>+</button>
      </div>
    </>
  );
};

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

Now the Todos component only re-renders when the todos that are passed to it through props are updated.

Styling React Using CSS

There are many ways to style React with CSS, this tutorial will take a closer look at three common ways:

- Inline styling
- CSS stylesheets
- CSS Modules

Inline Styling

To style an element with the inline style attribute, the value must be a JavaScript object:

Example:

Insert an object with the styling information:

```
const Header = () => {

  return (

    <>

      <h1 style={{color: "red"}}>Hello Style!</h1>

      <p>Add a little style!</p>

    </>

  );

}
```

```
import React from 'react';
import ReactDOM from 'react-dom/client';

const Header = () => {
  return (
    <>
      <h1 style={{color: "red"}}>Hello Style!</h1>
      <p>Add a little style!</p>
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header />);
```

Note: In JSX, JavaScript expressions are written inside curly braces, and since JavaScript objects also use curly braces, the styling in the example above is written inside two sets of curly braces `{{}}`.

camelCased Property Names

Since the inline CSS is written in a JavaScript object, properties with hyphen separators, like `background-color`, must be written with camel case syntax:

Example:

Use `backgroundColor` instead of `background-color`:

```
const Header = () => {

  return (

    <>

      <h1 style={{backgroundColor: "lightblue"}}>Hello Style!</h1>

      <p>Add a little style!</p>

    </>

  );

}
```

```
import React from 'react';
import ReactDOM from 'react-dom/client';

const Header = () => {
  return (
    <>
      <h1 style={{backgroundColor: "lightblue"}}>Hello Style!</h1>
      <p>Add a little style!</p>
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header />);
```

JavaScript Object

You can also create an object with styling information, and refer to it in the style attribute:

Example:

Create a style object named `myStyle`:

```
const Header = () => {

  const myStyle = {

    color: "white",

    backgroundColor: "DodgerBlue",

    padding: "10px",
    fontFamily: "Sans-Serif"

  };

  return (

    <>

      <h1 style={myStyle}>Hello Style!</h1>

      <p>Add a little style!</p>

    </>

  );

}
```

```
import React from 'react';
import ReactDOM from 'react-dom/client';

const Header = () => {
  const myStyle = {
    color: "white",
    backgroundColor: "DodgerBlue",
    padding: "10px",
    fontFamily: "Sans-Serif"
  };
  return (
    <>
      <h1 style={myStyle}>Hello Style!</h1>
      <p>Add a little style!</p>
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header />);
```

CSS Stylesheet

You can write your CSS styling in a separate file, just save the file with the `.css` file extension, and import it in your application.

App.css:

Create a new file called "App.css" and insert some CSS code in it:

```css
body {
  background-color: #282c34;
  color: white;
  padding: 40px;
  font-family: Sans-Serif;
  text-align: center;
}
```

Note: You can call the file whatever you like, just remember the correct file extension.

Import the stylesheet in your application:

index.js:

```jsx
import React from 'react';
import ReactDOM from 'react-dom/client';
import './App.css';

const Header = () => {
  return (
    <>
      <h1>Hello Style!</h1>
      <p>Add a little style!.</p>
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header />);
```

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './App.css';

const Header = () => {
  return (
    <>
      <h1>Hello Style!</h1>
      <p>Add a little style!.</p>
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header />);
```

CSS Modules

Another way of adding styles to your application is to use CSS Modules.

CSS Modules are convenient for components that are placed in separate files.

The CSS inside a module is available only for the component that imported it, and you do not have to worry about name conflicts.

Create the CSS module with the `.module.css` extension, example: `my-style.module.css`.

Create a new file called "my-style.module.css" and insert some CSS code in it:

my-style.module.css:

```
.bigblue {

  color: DodgerBlue;

  padding: 40px;

  font-family: Sans-Serif;

  text-align: center;

}
```

Import the stylesheet in your component:

Car.js:

```
import styles from './my-style.module.css';


const Car = () => {

  return <h1 className={styles.bigblue}>Hello Car!</h1>;

}


export default Car;
```

Import the component in your application:

index.js:

```
import ReactDOM from 'react-dom/client';

import Car from './Car.js';


const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Car />);
```

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import Car from './Car.js';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car />);
```

Styling React Using Sass

What is Sass

Sass is a CSS pre-processor.

Sass files are executed on the server and sends CSS to the browser.

You can learn more about Sass in our Sass Tutorial.

Can I use Sass?

If you use the `create-react-app` in your project, you can easily install and use Sass in your React projects.

Install Sass by running this command in your terminal:

```
npm i sass
```

Now you are ready to include Sass files in your project!

Create a Sass file

Create a Sass file the same way as you create CSS files, but Sass files have the file extension `.scss`

In Sass files you can use variables and other Sass functions:

Example

my-sass.scss:

Create a variable to define the color of the text:

```
$myColor: red;


h1 {
  color: $myColor;
}
```

Import the Sass file the same way as you imported a CSS file:

Example

index.js:

```
import React from 'react';

import ReactDOM from 'react-dom/client';

import './my-sass.scss';


const Header = () => {

  return (

    <>

      <h1>Hello Style!</h1>

      <p>Add a little style!.</p>

    </>

  );

}


const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Header />);
```

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './my-sass.scss';

const Header = () => {
  return (
    <>
      <h1>Hello Style!</h1>
      <p>Add a little style!.</p>
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header />);
```

React Hooks

Hooks were added to React in version 16.8.

Hooks allow function components to have access to state and other React features. Because of this, class components are generally no longer needed.

Although Hooks generally replace class components, there are no plans to remove classes from React.

What is a Hook?

Hooks allow us to "hook" into React features such as state and lifecycle methods.

Example:

Here is an example of a Hook. Don't worry if it doesn't make sense. We will go into more detail in the next section.

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

import ReactDOM from "react-dom/client";


function FavoriteColor() {

  const [color, setColor] = useState("red");


  return (

    <>

      <h1>My favorite color is {color}!</h1>

      <button

        type="button"

        onClick={() => setColor("blue")}
      >Blue</button>

      <button

        type="button"

        onClick={() => setColor("red")}

      >Red</button>

      <button

        type="button"

        onClick={() => setColor("pink")}

      >Pink</button>

      <button

        type="button"

        onClick={() => setColor("green")}

      >Green</button>

    </>

  );

}



const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<FavoriteColor />);
```

```
import React, { useState } from "react";
import ReactDOM from "react-dom/client";

function FavoriteColor() {
  const [color, setColor] = useState("red");

  return (
    <>
      <h1>My favorite color is {color}!</h1>
      <button
        type="button"
        onClick={() => setColor("blue")}
      >Blue</button>
      <button
        type="button"
        onClick={() => setColor("red")}
      >Red</button>
      <button
        type="button"
        onClick={() => setColor("pink")}
      >Pink</button>
      <button
        type="button"
        onClick={() => setColor("green")}
      >Green</button>
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<FavoriteColor />);
```

You must `import` Hooks from `react`.

Here we are using the `useState` Hook to keep track of the application state.

State generally refers to application data or properties that need to be tracked.

Hook Rules

There are 3 rules for hooks:

- Hooks can only be called inside React function components.
- Hooks can only be called at the top level of a component.
- Hooks cannot be conditional

Note: Hooks will not work in React class components.

Custom Hooks

If you have stateful logic that needs to be reused in several components, you can build your own custom Hooks.

React Custom Hooks

Hooks are reusable functions.

When you have component logic that needs to be used by multiple components, we can extract that logic to a custom Hook.

Custom Hooks start with "use". Example: `useFetch`.

Build a Hook

In the following code, we are fetching data in our `Home` component and displaying it.

We will use the [JSONPlaceholder]() service to fetch fake data. This service is great for testing applications when there is no existing data.

To learn more, check out the [JavaScript Fetch API]() section.

Use the JSONPlaceholder service to fetch fake "todo" items and display the titles on the page:

Example:

`index.js`:

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

import ReactDOM from "react-dom/client";


const Home = () => {

  const [data, setData] = useState(null);


  useEffect(() => {

    fetch("https://jsonplaceholder.typicode.com/todos")

      .then((res) => res.json())

      .then((data) => setData(data));

 }, []);
```

```
  return (

    <>

      {data &&

        data.map((item) => {

          return <p key={item.id}>{item.title}</p>;

        })}

    </>

  );

};



const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Home />);
```

```
import { useState, useEffect } from "react";
import ReactDOM from "react-dom/client";

const Home = () => {
  const [data, setData] = useState(null);

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

  return (
    <>
      {data &&
        data.map((item) => {
          return <p key={item.id}>{item.title}</p>;
        })}
    </>
  );
};

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Home />);
```

The fetch logic may be needed in other components as well, so we will extract that into a custom Hook.

Move the fetch logic to a new file to be used as a custom Hook:

Example:

useFetch.js:

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


const useFetch = (url) => {

  const [data, setData] = useState(null);


  useEffect(() => {

    fetch(url)

      .then((res) => res.json())

      .then((data) => setData(data));

  }, [url]);


  return [data];

};


export default useFetch;
```

index.js:

```
import ReactDOM from "react-dom/client";

import useFetch from "./useFetch";


const Home = () => {

  const [data] =
useFetch("https://jsonplaceholder.typicode.com/todos");


  return (
    <>

      {data &&

        data.map((item) => {

          return <p key={item.id}>{item.title}</p>;

        })}

    </>

  );

};



const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Home />);
```

```
import ReactDOM from "react-dom/client";
import useFetch from "./useFetch";

const Home = () => {
  const [data] =
useFetch("https://jsonplaceholder.typicode.com/todos");

  return (
    <>
      {data &&
        data.map((item) => {
          return <p key={item.id}>{item.title}</p>;
        })}
    </>
  );
};

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Home />);
```

Example Explained

We have created a new file called `useFetch.js` containing a function called `useFetch` which contains all of the logic needed to fetch our data.

We removed the hard-coded URL and replaced it with a `url` variable that can be passed to the custom Hook.

Lastly, we are returning our data from our Hook.

In `index.js`, we are importing our `useFetch` Hook and utilizing it like any other Hook. This is where we pass in the URL to fetch data from.

Now we can reuse this custom Hook in any component to fetch data from any URL.

Introduction to Redux More Redux and Client-Server Communication

Understanding React: React applications catch attention as the premier front-end JavaScript library in the advanced tech landscape. Companies, ranging from developing startups to industry giants, are gradually embracing this universal technology. Esteemed entities like Netflix, Airbnb, and The New York Times have integrated React into their websites and mobile applications.

Fundamentals of Redux: In the realm of JavaScript applications, Redux is a trustworthy state container. Handling an application's organization and data flow becomes increasingly demanding as it expands. This is absolutely where Redux steps in to deliver a solution.

Integrating React with Redux: This section put light on React with Redux and present a practical perception of the primary concepts linked to this widely adopted technology. To start, we will introduce ourselves to some fundamental Redux concepts. The next step is to develop a simple React application with Redux to compare it with one without it later. Through this exercise, the differences between the two accomplishments will become evident.

Before delving deeper into comprehending React Redux, let's first explore the rationale behind choosing Redux!

What is React?

React, more well-known as React.js or ReactJS, is an open-source JavaScript library for creating user interfaces or UI components, especially for single-page applications where users interact dynamically and require a persistent and flexible experience. In addition to Facebook, there is a community of developers and companies that maintain it.

What is Redux?

Redux supports a trustworthy state container intended for JavaScript applications. Its objective is to facilitate the development of applications that reveal persistent behavior, operate flawlessly across various environments (including client, server, and native platforms), and are easily testable.

Redux organizes an application's state at its core through a singular global object known as the Store. Redux flawlessly integrates with any JavaScript framework or library as an adaptable state management tool. By storing the application's state, Redux permits components to access this state smoothly through a committed state store.

Why Redux?

Transferring states between components in React can be difficult, especially when holding track of the source component becomes complicated. This intricacy develops, especially when dealing with multiple states within a substantial application. Redux addresses this issue by securing all states in a focused repository known as a store. This method streamlines the management and transfer of states, as they are appropriately stored in an integrated location. Accordingly, every component in the application benefits from direct access to the necessary state directly from the store, streamlining the state handling process.

When to use Redux?

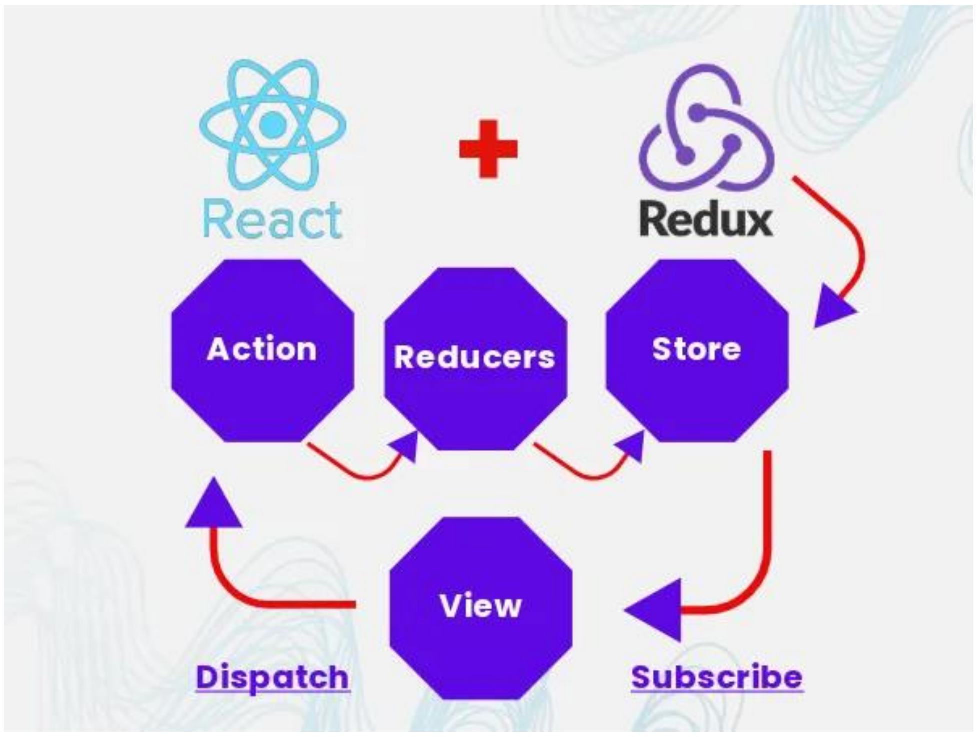
React Redux becomes essential in developing heavy applications that highlight multiple components. It's critical when these components need to access an application state.

Due to one-way data binding in the React framework, related elements cannot directly access each other's states. To conquer this limitation, the practice of elevating the state is employed, where a parent element holds the state for its sub-element to utilize.

While this technique is sufficient for simple "hello world" applications, it becomes awkward in the context of real-world applications. Handling the passing of states and methods becomes a challenging task. React Redux addresses this challenge by providing an efficient mechanism for approaching the global state from any component.

In the abstract, Redux is essential when forming significant applications with various states. It is a global state management library, facilitating persistent state sharing among components.

How Redux Works?



Redux is an essential application. The whole state of the application is kept in a central store. Each component can be available in a stored state without passing down props from one component to the next. There are three categories of building blocks: actions, stores, and reducers. Let's have a quick look at each of them. This is important as it helps you grasp the benefits of React Redux and how to utilize it. We'll use Redux to build a similar example to the login component above.

Actions in Redux

Simply put, actions are events. They are the only way to transfer data from your application to your Redux store. The data might come from user interactions, API requests, or even form submissions.

The store.dispatch() function is used to dispatch actions. Actions are simple JavaScript objects that must include a type property to specify the action to be performed. They must also have a payload containing the information that the action should work on. An action maker is used to create action.

Here's an example to explain the action during login:

```
{
type: "LOGIN",
payload: {
username: "foo",
password: "bar"
}
}
Here is an instance of its action creator:
const set LoginStatus = (name, password) => {
return {
type: "LOGIN",
payload: {
username: "foo",
password: "bar"
}
}
}
```

Reducers in Redux

Reducers take the applications current state, execute an action, and return a new form. These states are objects that describe how an application's state changes in response to an action delivered to the store.

It is based on JavaScript's reduction function, which calculates a single value from several values after executing a callback function.

Here is an example to show reducers working in Redux:

```
const LoginComponent = (state = initialState, action)
=>
{
switch (action.type)
{
// This reducer control any action with type "LOGIN"
case "LOGIN":
return state.map(user => {
if (user.username !== action.username) {
return user;
}
if (user.password == action.password) {
return {
...user,
login_status: "LOGGED IN"
}
}
});
default:
return state;
}
};
```

Store in Redux

The application state is stored in the store. In any Redux application, it is strongly advised to retain only one store. You can use helper methods to retrieve the saved state, change it, and register or unregister listeners.
Let’s make a store for our login app:

```
const store = createStore(LoginComponent);
```

Actions on the state always lead to a new state. Thus the state is relatively predictable and straightforward.
Now that we’ve understood a bit more about Redux let’s go to our earlier login component example and see how Redux can enhance it.

```
class App extends React.Component {
render() {
return (
<div>
<Status user={this.props.user.name}/>
<Login login={this.props.setLoginStatus}/>
</div>
)
}
}
```

With Redux, there is just one general state in the store, and each component has access to it. This reduces the need to transfer form from one component to another continuously. You can also choose a slice from the store for a particular component, making your app more efficient.
Redux middleware: Developers can use Redux in React JS to capture all the actions dispatched by components. This is done before they are delivered to the reducer function. This interception is proficient through the use of middleware.
Taking the Login component from the preceding section as a model, we can purify the user’s input before sending it to our store for additional processing. This is possible using the Redux middleware. Technically, the middleware calls the following method after processing the current action. This method was received in an argument after processing the present action. These are named after each dispatch.
Here’s an example of a basic middleware:

```
function simpleMiddleware({ getState, dispatch }) {
return function(next){
return function(action){
// processing
const nextAction = next(action);
// read the next state
const state = getState();
// return the next action or you can dispatch any other
action
return nextAction;
}
}
}
```
This may appear to be a difficult challenge. Still, in most situations, you may not need to develop your middleware since the large Redux community has already made a number of them available. If you believe middleware is necessary, you will appreciate it since it provides you with the power to perform excellent work with the most acceptable abstraction.

Why use Redux?

Redux is a self-contained library consistent with several UI layers or frameworks such as React, Angular, Vue, Ember, and vanilla JS. Despite the common association of Redux with React, it functions independently of any specific UI framework.

When integrating Redux into a UI framework, the typical approach involves using a "UI binding" library to establish a connection between Redux and the selected UI framework. This includes preventing direct interaction with the store from the UI code.

Created as the official Redux UI binding library for React, React Redux fulfills this role when integrating Redux with React. If your project includes both Redux and React, combining React and Redux to flawlessly link these two libraries is wise.

To understand the consequences of using React Redux, it is beneficial to grasp the role of a "UI binding library" and how it promotes the integration of Redux with the user interface.

Redux makes the State Predictable

Redux ensures a constant state. Because reducers are pure functions, the same result is always produced if the same state and action are provided. The status is also persistent and never changes. This allows complex tasks such as infinite undo and redo to be implemented. It is also feasible to implement time travel, allowing the capability to navigate back and forth between previous states and observe the effects in real-time.

Redux is Maintainable: Redux strengthens operability in software development by providing a focused and anticipated state management system. It accomplishes this by storing the application's state in a single, systematic data store called the "store." With a clear separation of concerns, Redux ensures that data flows, simplifying the debugging and tracking of changes.

This focused method promotes simpler debugging and testing, as the entire application state is available in one location. Actions that symbolize state changes are transmitted to the store, making it straightforward to follow and understand how the application evolves.

Further, Redux promotes usability through its firmness principle. Instead of changing the existing state, Redux generates new states with each action, preventing unintentional side effects and making rolling back or replaying actions simpler.

Debugging is easy in Redux: Debugging applications is simplified with React Redux. Understanding code mistakes, network failures, and other bugs during production is simple by recording actions and states. It also offers excellent DevTools that enable you to time-travel activities, persist actions on page refresh, and much more. Debugging takes longer than developing features in medium and large-scale programs. Redux DevTools makes it easier to take advantage of all that Redux offers.

Performance Benefits: Keeping the app's state global would degrade performance. React Redux provides several performance optimizations internally, ensuring that your linked component renders only when necessary.

Server-side Rendering: Redux can also be helpful for server-side rendering, commonly known as SSR. It allows you to manage the app's first render by providing its state and response to the server request. The required components are subsequently produced in HTML and delivered to the customers.

Getting Started with Redux

Redux is a JS library for predictable and maintainable global state management.

It helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test. On top of that, it provides a great developer experience, such as live code editing combined with a time traveling debugger.

You can use Redux together with React, or with any other view library. It is tiny (2kB, including dependencies), but has a large ecosystem of addons available.

Redux Toolkit is our official recommended approach for writing Redux logic. It wraps around the Redux core, and contains packages and functions that we think are essential for building a Redux app. Redux Toolkit builds in our suggested best practices, simplifies most Redux tasks, prevents common mistakes, and makes it easier to write Redux applications.

RTK includes utilities that help simplify many common use cases, including store setup, creating reducers and writing immutable update logic, and even creating entire "slices" of state at once.

Whether you're a brand new Redux user setting up your first project, or an experienced user who wants to simplify an existing application, **Redux Toolkit** can help you make your Redux code better.

Installation Redux Toolkit

Redux Toolkit is available as a package on NPM for use with a module bundler or in a Node application:

```
# NPM
npm install @reduxjs/toolkit

# Yarn
yarn add @reduxjs/toolkit
```

Create a React Redux App

The recommended way to start new apps with React and Redux is by using our official Redux+TS template for Vite, or by creating a new Next.js project using Next's `with-redux` template.

Both of these already have Redux Toolkit and React-Redux configured appropriately for that build tool, and come with a small example app that demonstrates how to use several of Redux Toolkit's features.

```
# Vite with our Redux+TS template
# (using the `degit` tool to clone and extract the template)
npx degit reduxjs/redux-templates/packages/vite-template-redux my-app

# Next.js using the `with-redux` template
npx create-next-app --example with-redux my-app
```

We do not currently have official React Native templates, but recommend these templates for standard React Native and for Expo:

- https://github.com/rahsheen/react-native-template-redux-typescript
- https://github.com/rahsheen/expo-template-redux-typescript

Redux Core

The Redux core library is available as a package on NPM for use with a module bundler or in a Node application:

```
# NPM
npm install redux

# Yarn
yarn add redux
```

The package includes a precompiled ESM build that can be used as a `<script type="module">` tag directly in the browser.

For more details, see the Installation page.

Basic Example

The whole global state of your app is stored in an object tree inside a single *store*. The only way to change the state tree is to create an *action*, an object describing what happened, and *dispatch* it to the store. To specify how state gets updated in response to an action, you write pure *reducer* functions that calculate a new state based on the old state and the action.

Redux Toolkit simplifies the process of writing Redux logic and setting up the store. With Redux Toolkit, the basic app logic looks like:

```
import { createSlice, configureStore } from '@reduxjs/toolkit'

const counterSlice = createSlice({
  name: 'counter',
  initialState: {
    value: 0
  },
  reducers: {
    incremented: state => {
      // Redux Toolkit allows us to write "mutating" logic in reducers. It
      // doesn't actually mutate the state because it uses the Immer
library,
      // which detects changes to a "draft state" and produces a brand new
      // immutable state based off those changes
      state.value += 1
    },
    decremented: state => {
      state.value -= 1
    }
  }
})

export const { incremented, decremented } = counterSlice.actions

const store = configureStore({
  reducer: counterSlice.reducer
})

// Can still subscribe to the store
store.subscribe(() => console.log(store.getState()))

// Still pass action objects to `dispatch`, but they're created for us
store.dispatch(incremented())
// {value: 1}
store.dispatch(incremented())
// {value: 2}
store.dispatch(decremented())
// {value: 1}
```

Instead of mutating the state directly, you specify the mutations you want to happen with plain objects called *actions*. Then you write a special function called a *reducer* to decide how every action transforms the entire application's state.

In a typical Redux app, there is just a single store with a single root reducer function. As your app grows, you split the root reducer into smaller reducers independently operating on the different parts of the state tree. This is exactly like how there is just one root component in a React app, but it is composed out of many small components.

This architecture might seem like a lot for a counter app, but the beauty of this pattern is how well it scales to large and complex apps. It also enables very powerful developer tools, because it is possible to trace every mutation to the action that caused it. You can record user sessions and reproduce them just by replaying every action.

Redux Toolkit allows us to write shorter logic that's easier to read, while still following the same Redux behavior and data flow.

Legacy Example

For comparison, the original Redux legacy syntax (with no abstractions) looks like this:

```
import { createStore } from 'redux'

/**
 * This is a reducer - a function that takes a current state value and an
 * action object describing "what happened", and returns a new state value.
 * A reducer's function signature is: (state, action) => newState
 *
 * The Redux state should contain only plain JS objects, arrays, and
primitives.
 * The root state value is usually an object. It's important that you
should
 * not mutate the state object, but return a new object if the state
changes.
 *
 * You can use any conditional logic you want in a reducer. In this
example,
 * we use a switch statement, but it's not required.
 */
function counterReducer(state = { value: 0 }, action) {
  switch (action.type) {
    case 'counter/incremented':
      return { value: state.value + 1 }
    case 'counter/decremented':
      return { value: state.value - 1 }
    default:
      return state
  }
}

// Create a Redux store holding the state of your app.
// Its API is { subscribe, dispatch, getState }.
let store = createStore(counterReducer)

// You can use subscribe() to update the UI in response to state changes.
// Normally you'd use a view binding library (e.g. React Redux) rather than
subscribe() directly.
// There may be additional use cases where it's helpful to subscribe as
well.

store.subscribe(() => console.log(store.getState()))

// The only way to mutate the internal state is to dispatch an action.
// The actions can be serialized, logged or stored and later replayed.
store.dispatch({ type: 'counter/incremented' })
// {value: 1}
store.dispatch({ type: 'counter/incremented' })
```

```
// {value: 2}
store.dispatch({ type: 'counter/decremented' })
// {value: 1}
```

Steps To Implement React-Redux

Step 1: Setting Up the Project

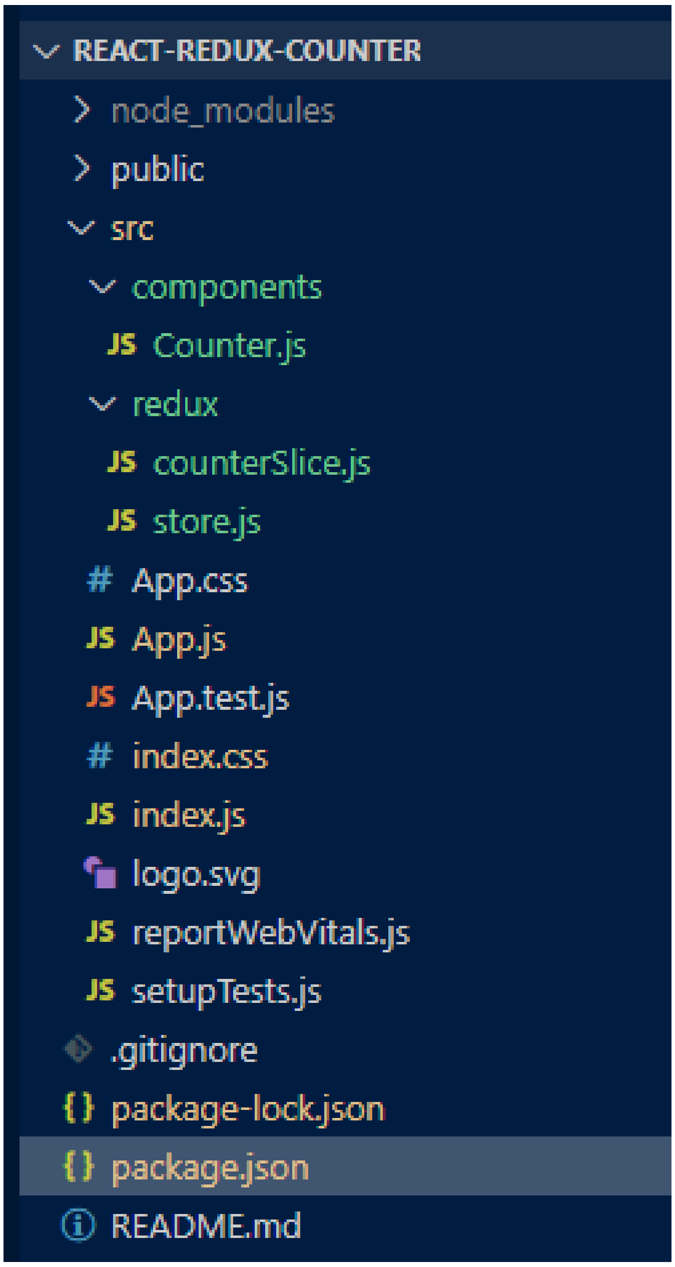
First, create a new React app using create-react-app:

```
npx create-react-app react-redux-counter
cd react-redux-counter
```

Next, install redux and react-redux:

```
npm install redux react-redux
```

Folder Structure



```
REACT-REDUX-COUNTER
  node_modules
  public
  src
    components
      Counter.js
    redux
      counterSlice.js
      store.js
    App.css
    App.js
    App.test.js
    index.css
    index.js
    logo.svg
    reportWebVitals.js
    setupTests.js
  .gitignore
  package-lock.json
  package.json
  README.md
```

Folder Structure

Dependencies

```
"dependencies": {
    "@testing-library/jest-dom": "^5.17.0",
    "@testing-library/react": "^13.4.0",
    "@testing-library/user-event": "^13.5.0",
    "react": "^18.3.1",
    "react-dom": "^18.3.1",
    "react-redux": "^9.1.2",
    "react-scripts": "5.0.1",
    "redux": "^5.0.1",
    "web-vitals": "^2.1.4"
  }
```

Step 2: Create the Redux Slice (counterSlice.js)

In the src/redux/ folder, create a file called counterSlice.js. This will define the actions and reducer for the counter.

JavaScript

```javascript
// src/redux/counterSlice.js

const initialState = {
    count: 0,
};

// Reducer function that updates the state based on actions
export default function counterReducer(state = initialState, action) {
    switch (action.type) {
        case 'INCREMENT':
            return { ...state, count: state.count + 1 };
        case 'DECREMENT':
            return { ...state, count: state.count - 1 };
        default:
            return state;
    }
}

// Action creators
export const increment = () => ({ type: 'INCREMENT' });
export const decrement = () => ({ type: 'DECREMENT' });
```

Step 3: Create the Redux Store (store.js)

Now, create the Redux store in the src/redux/store.js file:

JavaScript

```javascript
// src/redux/store.js

import { createStore } from 'redux';
import counterReducer from './counterSlice';

// Create a Redux store holding the state of the counter
const store = createStore(counterReducer);

export default store;
```

Step 4: Create the Counter Component (Counter.js)

Now, create a simple Counter component in the src/components/Counter.js file. This component will access the state and dispatch actions to increment or decrement the count.

JavaScript

```javascript
// src/components/Counter.js

import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { increment, decrement } from '../redux/counterSlice';

const Counter = () => {
    // Access the count value from the Redux store
    const count = useSelector((state) => state.count);
    // Get the dispatch function from Redux
    const dispatch = useDispatch();

    return (
        <div>
            <h1>Count: {count}</h1>
            <button onClick={() => dispatch(increment())}>Increment</button>
            <button onClick={() => dispatch(decrement())}>Decrement</button>
        </div>
    );
};

export default Counter;
```

Step 5: Connect Redux to React

Next, wrap your App component in a Redux Provider so that the store is available throughout the app.

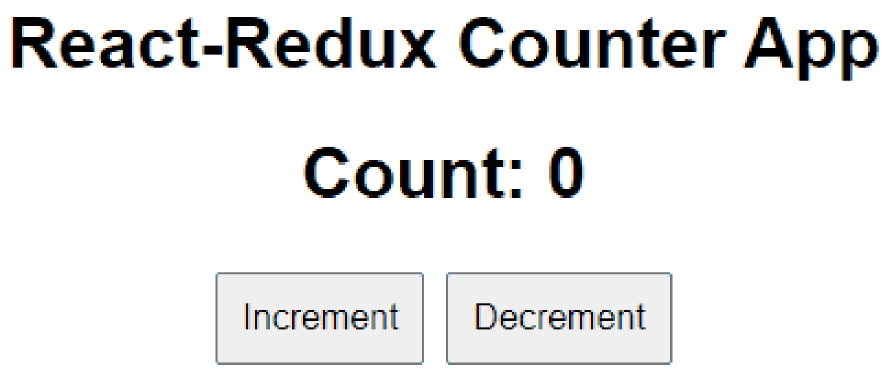
CSSJavaScriptJavaScript

```css
/* src/index.css */

body {
    font-family: Arial, sans-serif;
    text-align: center;
}

button {
    margin: 5px;
    padding: 10px;
    font-size: 16px;
}
```

To start the application run the following command.

```
npm start
```

Output



Introduction to React-Redux