**React**

Conditional rendering

List and key

Basic form handling

React router:

Programmatic navigation

Setup json server

Api axios call:

SOCKET IO In react:

React crud operations:

React search filter

Local storage:

**Conditional rendering:** there are two type of

1) logical operator using &&

2) ternary operator

## 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} />);

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

## Ternary Operator

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

condition ? true : false

### 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} />);

**React List and Key:**

**List:**

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](https://www.w3schools.com/react/react_es6.asp).

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 />);

## 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 />);

**Basic form handling in React:**

### Example 1:

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 />);

**Example 2:**

## 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 />);

## 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 />);

**React router:**

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 />);

## 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.

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;

## CSS Styling in react:

## 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>

</>

);

}

### 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>

</>

);

}

### 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>

</>

);

}

### Introduction

Pages on modern websites, and especially **Single-Page Applications (SPAs)** do not follow the traditional method of loading new pages completely whenever something changes on them. **Client-Side Routing** is used instead - to route towards resources to load another part of the program or structurally change the application's entire view if needed, when an action (such as clicking a button or link) is performed.

React is a massively popular library for front-end development used to create highly responsive user interfaces, and naturally, it has its own Router, which performs Client-Side Routing - the react-router-dom.

*In this guide, we'll take a look at how to create routes in a React application, programmatically navigate between routes, as well as send and retrieve data between them, using React Router.*

### Creating a React Application

Let's start out by creating a simple React application via the command line:

$ npx create-react-app router-sample

Once created, let's move into the project's directory, and start the application:

$ cd router-sample

$ npm start

This will start up a server on localhost:3000 and your default browser will fire up to serve the application. Before creating any new files to serve on this endpoint, let's install react-router-dom, since it doesn't come pre-packaged.

#### **Installing React Router**

As usual, installing a package using npm is as simple as running a single command:

$ npm install react-router-dom

#### **Creating New Routes with React Router**

The react-router-dom package makes it simple to create new routes. To begin, you wrap the entire application with the <BrowserRouter> tag. We do this to gain access to the browser's history object. Then you define your router links, as well as the components that will be used for each route.

To demonstrate this, let's create a new file called About.js in the /src folder:

const About = () => {

  return (

    <div>

      <h1>About page here!</h1>

      <p>

        Lorem ipsum dolor sit amet consectetur adipisicing elit. Fugit, modi!

      </p>

    </div>

  );

};

export default About;

Now, let's update the src/index.js page and import About from the file we've just created. Within the <BrowserRouter> tag, we'll define our routes and components associated with them:

import { render } from "react-dom";

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

import App from "./App";

import About from "./About";

render(

  <BrowserRouter>

    <Routes>

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

      <Route path="about" element={<About />} />

    </Routes>

  </BrowserRouter>,

  document.getElementById("root")

);

We've imported the <BrowserRouter> here, and we'd wrapped our entire application around it. We'd also selected App.js as the component for our home page (under the / endpoint), and About.js as the component for the /about page.

Finally, let's adapt the App.js file, which will, again, be the main entry point for the application and serve our home page:

import { Link } from "react-router-dom";

function App() {

  return (

    <div className="App">

      <h1>Welcome to my react app!</h1>

      <p>

        Lorem ipsum dolor sit amet consectetur adipisicing elit. Accusamus,

        pariatur?

      </p>

      <br />

      <Link to="/about">About Page</Link>

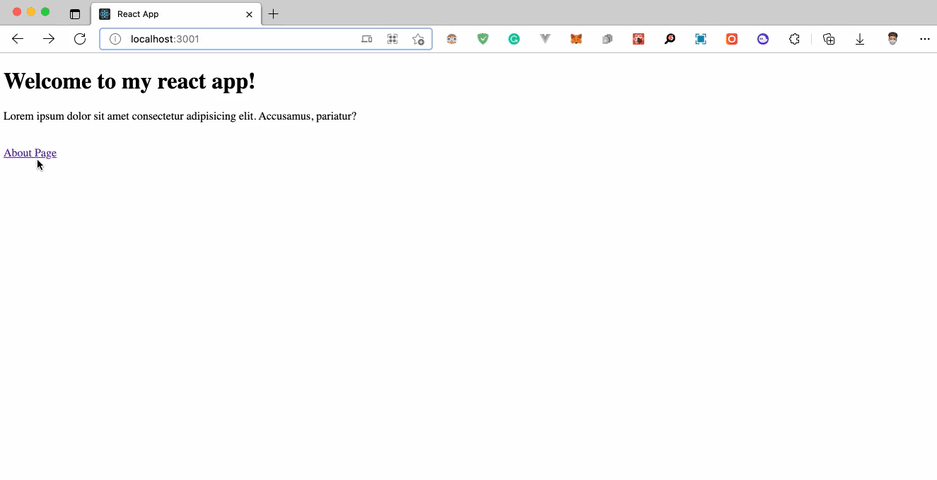
    </div>

  );

}

export default App;

At this stage, we've created two routes: the entry route (/) and the /about route, and we should be able to easily navigate from the home page to the about page when we visit our application:



### React Router Hooks (Navigation Methods)

Hooks were added to React recently, which are functions that allow you to "hook" into the application's state, without having to write new classes. The React Router comes with a few hooks that allow us to programmatically access the Router State and navigate between components. Among these are the following hooks:

* useHistory()
* userNavigate()
* useLocation()
* useParams()

To use any of these hooks, we must first import them from the react-router-dom package and then specify a variable that invokes the hook; we'll go through this in more detail in the sections that follow.

#### **useHistory()**

**Note:** The useHistory() hook has been deprecated in the latest version of React Router. If you're using React Router V6, you'll want to use the useNavigate() hook instead. It's covered right after useHistory().

The useHistory() hook provides direct access to React Router's history instances, enabling us to perform actions like retrieving the number of entries in the history stack, adding, altering, or removing an entry from the stack, and so on.

Some of the most useful methods here include:

* goBack() – Go backward in history.
* goForward() - Go forward in history.
* push() - Add a new entry to the history stack, i.e., navigate to a new URL.
* replace() - Similar to push() in that it replaces the current stack in the history, but unlike push(), the user cannot travel back in history, i.e. clicking the browser back button will not return to the previous state.

With just this, we can navigate from our About page to the home page programmatically, by push()ing a new entry into the history, effectively navigating the user to the new entry:

import React from "react";

import { useHistory } from "react-router-dom";

const About = () => {

  let history = useHistory();

  const goHome = () => {

    history.push("/");

  };

  return (

    <div>

      <h1>About page here!</h1>

      <p>

        Lorem ipsum dolor sit amet consectetur adipisicing elit. Fugit, modi!

      </p>

      <button onClick={goHome}>Go to home page</button>

    </div>

  );

};

export default About;

Here, we just import the useHistory() hook and create a new goHome() function that executes on a button click. It's just a wrapper for a push() call.

We can also transfer the state or send arbitrary data to the route we are navigating to. We can do this by sending an object to the push methods instead of the pathname as a string, and adding our state to a distinct object:

history.push({

  pathname: '/blog,

  search: '?blogId=12,  // Query string

  state: {              // Location state

    author\_name: "John Doe",

  },

});

After covering useNavigate(), we'll take a look at how to access this data using the useLocation() and useParams() hooks.

#### **useNavigate()**

If you're using the most recent version of React Router, the useHistory() hook has been deprecated in favor of useNavigate(). The approach is nearly identical; the main difference is that the useNavigate() hook does not accept methods like .push() or .replace(). You just navigate() to a certain entry:

import React from "react";

import { useNavigate } from "react-router-dom";

const About = () => {

  let navigate = useNavigate();

  const goHome = () => {

    navigate("/");

  };

  return (

    <div>

      <h1>About page here!</h1>

      <p>

        Lorem ipsum dolor sit amet consectetur adipisicing elit. Fugit, modi!

      </p>

      <button onClick={goHome}>Go to home page</button>

    </div>

  );

};

export default About;

And, of course, this method also allows us to transfer states between routes:

navigate("/blog", { state: { author\_name: "John Doe" } });

**Retrieving Data Between Routes with useLocation() and useParams()**

The useLocation() hook provides us access to the browser's location object. Consider it the hook in charge of obtaining the necessary information about the current route:

import { useNavigate, useLocation } from "react-router-dom";

/\*...\*/

let location = useLocation();

console.log(location);

Running the code above should return all of the information about the current route, as shown below:

{

  "pathname": "/about",

  "search": "",

  "hash": "",

  "state": null,

  "key": "default"

}

It's reasonable to think of the useLocation() hook as a useState() hook that updates the state to a new location whenever the URL changes. On the other hand, the userParams() hook can be used to get the value of URL parameters. When called, useParams() provides an object that maps the names of URL parameters to their values in the current URL.

For example, in our router configuration, say we've had a dynamic route:

<Route path="/about/:user\_id">

 <About />

</Route>

And on another page, we have a link component that points to some information pertaining to User 2:

<Link to="/about/2">About User 2</Link>

When parameters are passed, like we've seen before - we can access the parameters via the userParams() hook:

import { useParams } from "react-router-dom";

const About = () => {

  const { user\_id } = useParams();

  return (

    <div>

      <h1>About user {user\_id}</h1>

      <p>

        Lorem ipsum dolor...

      </p>

    </div>

  );

};

export default About;

### Conclusion

React Router is used for Client-Side Routing of resources. In this guide, we've taken a quick look at how you can programmatically navigate an application using React Router as well as how to transfer states between routes when you do.

**What is Programmatic Navigation?**

Programmatic Navigation refers to when a user is redirected as a result of an action that occurs on a route. A login or signup action or form submission action on a route is a typical example of navigating programmatically. In this article, we’ll look at a myriad of approaches to navigating programmatically with React Router.

## Using Redirect Component

The primary way you programmatically navigate using React Router v4+ is by using a <Redirect /> component, and it’s a recommended method that helps the user navigate between routes.

Using the Redirect component is a different approach but just as valid. The idea is to have it pointing at a state in the component, and if that condition is fulfilled, then navigate.

Some might argue that this method requires more work as one needs to create a new prop on the component’s state and add a condition to the render method to check when to render the Redirect component. This is a fact, but a counter and valid argument, from those who prefer explicit to implicit: It points to the idea that explicitly defining and modifying your state is better as it makes the code more readable against the implicit state handled by an imperative API such as history.push, which we will go over in a bit.

Here’s a code example of how to use the Redirect component.

Codesandbox: <https://codesandbox.io/s/gallant-meitner-bshng?file=/src/App.js>

import React, { useState } from 'react';

import { Redirect } from 'react-router-dom';

import { userLogin } from './userAction';

import Form from './Form';

const Login = () => {

const [isLoggedIn, setIsLoggedIn] = useState(false);

const handleLogin = async (userDetail) => {

const success = await userLogin(userDetail);

if(success) setIsLoggedIn(true);

}

if (isLoggedIn) {

return <Redirect to='/profile' />

}

return (

<>

<h1>Login</h1>

<Form onSubmit={handleLogin} />

</>

)

}

export default Login;

JavaScript

## Using history.push() Method

history.push() is another approach where we make use of the history props React Router provides while rendering a component.

In other words, this works when the component is being rendered by React Router, bypassing the component as a Component prop to a Route. If this is the case, the React Router exposes three props to the component: location, match and history.

For another example of using history.push() with React components, see this [demo of the KendoReact Drawer component](https://www.telerik.com/kendo-react-ui/components/layout/drawer/routing/).

We’ll focus on the history prop. The history prop keeps track of all the session history under the hood and provides us with different methods to manipulate it.

The push method is essential and is used to push a path as a route to the history [stack](https://www.telerik.com/blogs/stack-queue-javascript), which executes as Last In First Out (LIFO). This causes the app to redirect to the last route added, thereby redirecting the user to a specified route. The example below assumes the component is rendered with React Router.

Codesandbox: <https://codesandbox.io/s/angry-saha-djh3z?file=/src/App.js>

import React from "react";

import { userLogin } from "./userAction";

import Form from "./Form";

const Login = props => {

const handleLogin = async userDetail => {

const success = await userLogin(userDetail);

if (success) props.history.push("/profile");

};

return (

<>

<h1>Login</h1>

<Form onSubmit={handleLogin} />

</>

);

};

export default Login;

JavaScript

## Using withRouter Method

We mentioned earlier that for a component to have access props.history.push it must have been rendered with React Router. There are cases where this might not be the case. Thus, we render a component ourselves. To make the history property available to the component, the React Router team created the Higher Order Component (HOC) withRouter. Wrapping a component with this HOC exposes the properties as well.

Codesandbox: <https://codesandbox.io/s/silent-rain-l19lg?file=/src/App.js:0-442>

import React from 'react';

import { withRouter } from 'react-router-dom';

import { userLogin } from './userAction';

import Form from './Form';

const Login = (props) => {

const handleLogin = async (userDetail) => {

const success = await userLogin(userDetail);

if(success) props.history.push('/profile');

}

return (

<>

<h1>Login</h1>

<Form onSubmit={handleLogin} />

</>

)

}

export default withRouter(Login);

JavaScript

## Using useHistory Hook

As of recent versions of React Router (v5.1) and React (v16.8), we have a new method called the useHistory hook which embraces the power of React Hooks. This is used for programmatic navigation purposes within a functional component. The useHistory hook gives you access to the [history](https://reacttraining.com/react-router/web/api/history) instance that we can use to navigate between pages, whether the component has been rendered by React Router or not, and this eliminates the need for using withRouter.

Codesandbox: <https://codesandbox.io/s/serene-cookies-hc629?file=/src/App.js>

import { useHistory } from "react-router-dom";

const HomeButton = () =>{

let history = useHistory();

const handleClick = () => {

history.push("/home");

}

return (

<button type="button" onClick={handleClick}>

Go home

</button>

);

}

export default HomeButton;

JavaScript

## Conclusion

The main focus of this article was to share how you can safely navigate between components using the React Router package.

Considering React has a declarative approach to building UIs, using Redirect is the recommended approach for navigation when the Link cannot be used. There is no harm in using the other methods as they are all supported and semantically correct.

Also, with the introduction of useHistory together other other APIs in the 5.1.2 release, it becomes even easier to navigate programmatically as long as you understand how to use React Hooks

**Axios Api calls:**

***Example 1:***

import React from "react";

import axios from "axios";

class App extends React.Component {

state = {

  newfiles: null,

};

handleFile(e) {

  // Getting the files from the input

  let newfiles = e.target.newfiles;

  this.setState({ newfiles });

}

handleUpload(e) {

  let newfiles = this.state.newfiles;

  let formData = new FormData();

  // Adding files to the formdata

  formData.append("image", newfiles);

  formData.append("name", "Name");

  axios({

  // Endpoint to send files

  url: "http://localhost:8080/files",

  method: "POST",

  headers: {

    // Add any auth token here

    authorization: "your token comes here",

  },

  // Attaching the form data

  data: formData,

  })

  // Handle the response from backend here

  .then((res) => { })

  // Catch errors if any

  .catch((err) => { });

}

render() {

  return (

  <div>

    <h1>Select your files</h1>

    <input

    type="file"

    // To select multiple files

    multiple="multiple"

    onChange={(e) => this.handleFile(e)}

    />

    <button onClick={(e) => this.handleUpload(e)}>

    Send Files

    </button>

  </div>

  );

}

}

export default App;

**Example 2: get call**

const getCustomersData = () => {

  axios

  .get("https://jsonplaceholder.typicode.com/customers")

  .then(data => console.log(data.data))

  .catch(error => console.log(error));

  };

  getCustomersData();

**Shorthand Methods in Axios:**Below are some shorthand methods of Axios…

* axios.request(config)
* axios.head(url[, config])
* axios.get(url[, config])
* axios.post(url[, data[, config]])
* axios.put(url[, data[, config]])
* axios.delete(url[, config])
* axios.options(url[, config])
* axios.patch(url[, data[, config]])

**Delete Request With Axios:**To send the delete request to the server axios.delete is used. You need to specify two parameters while making this request URL and optional config.

axios.delete(url, {

  data: { foo: "bar" },

  headers: { "Authorization": "\*\*\*\*\*\*" }

});

While sending the delete request you will have to set the request body and headers. Use config.data for this purpose. In the above POST request, modify the code as given below…

handleSubmit = event => {

  event.preventDefault();

  axios.delete(

  `https://jsonplaceholder.typicode.com/posts/${this.state.postName}`)

  .then(res => {

  console.log(res);

  console.log(res.data);

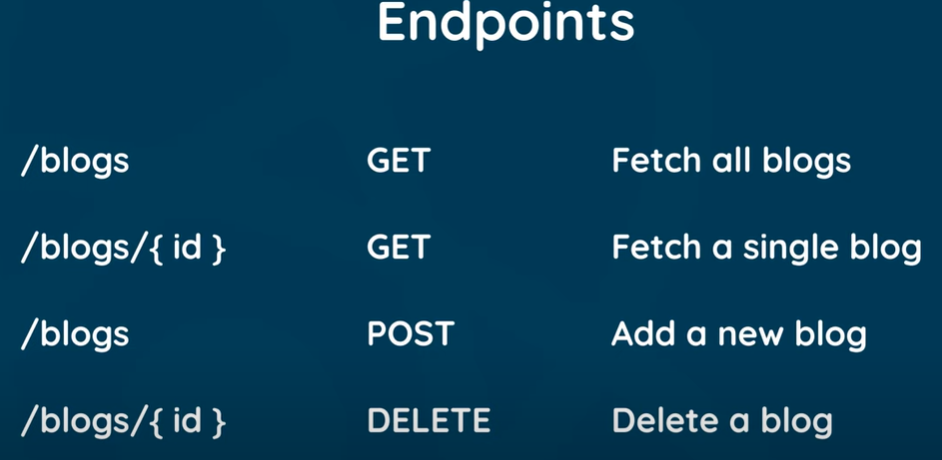
  })

  }

**Setup json server :**

**Command to run the json data on the port number 8000***:* npx json-server - - watch data/db.json - - port 8000

Endpoints:

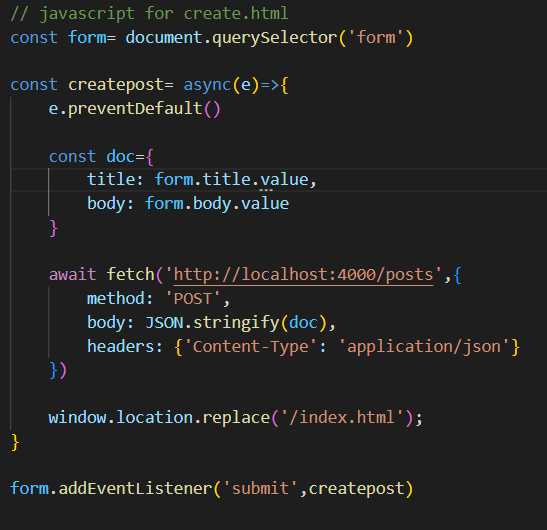


In Javascript installation and running process:

**npm install -g json-server**

**json-server --watch db.json**

**Example => create.js**



**details.js**



**Index.js**



Refer to the GIT HUB (saitejarapelli154@gmail.com)

**Here's how you can use React hooks with Socket.IO:**

Here's how you can use React hooks with Socket.IO:

**Example:**

import React, { useState, useEffect } from 'react';  
import io from 'socket.io-client';  
  
const socket = io();  
  
function App() {  
 const [isConnected, setIsConnected] = useState(socket.connected);  
 const [lastPong, setLastPong] = useState(null);  
  
 useEffect(() => {  
 socket.on('connect', () => {  
 setIsConnected(true);  
 });  
  
 socket.on('disconnect', () => {  
 setIsConnected(false);  
 });  
  
 socket.on('pong', () => {  
 setLastPong(new Date().toISOString());  
 });  
  
 return () => {  
 socket.off('connect');  
 socket.off('disconnect');  
 socket.off('pong');  
 };  
 }, []);  
  
 const sendPing = () => {  
 socket.emit('ping');  
 }  
  
 return (  
 <div>  
 <p>Connected: { '' + isConnected }</p>  
 <p>Last pong: { lastPong || '-' }</p>  
 <button onClick={ sendPing }>Send ping</button>  
 </div>  
 );  
}  
  
export default App;

Notes:

* the 2nd argument of the useEffect() method must be [], or else the hook will be triggered every time a new message arrives

useEffect(() => {  
 *// ...*  
}, []);

Copy

* the listeners must be removed in the cleanup step, in order to prevent multiple event registrations

useEffect(() => {  
 *// ...*  
 return () => {  
 socket.off('connect');  
 socket.off('disconnect');  
 socket.off('pong');  
 };  
}, []);

**Fetch data using useEffect:**



**How to Implement localStorage in React**

localStorage provides us with access to a browser's storage object, which includes five methods:

* setItem(): This method is used to add a key and a value to localStorage.
* getItem(): This method is used to get an item from localStorage using the key.
* removeItem(): This technique is used to delete an item from localStorage based on its key.
* clear(): This technique is used to delete all instances of localStorage.
* key(): When you supply a number, it aids in the retrieval of a localStorage key.

In this post, we will only consider the most popular methods, which are the first two methods.

**How to Use the**setItem()**Method**

By giving values to a key, this technique is used to store objects in localStorage. This value can be of any datatype, including text, integer, object, array, and so on.

It is vital to remember that in order to store data in localStorage, you must first stringify it with the JSON.stringify() function.

const [items, setItems] = useState([]);

useEffect(() => {

  localStorage.setItem('items', JSON.stringify(items));

}, [items]);

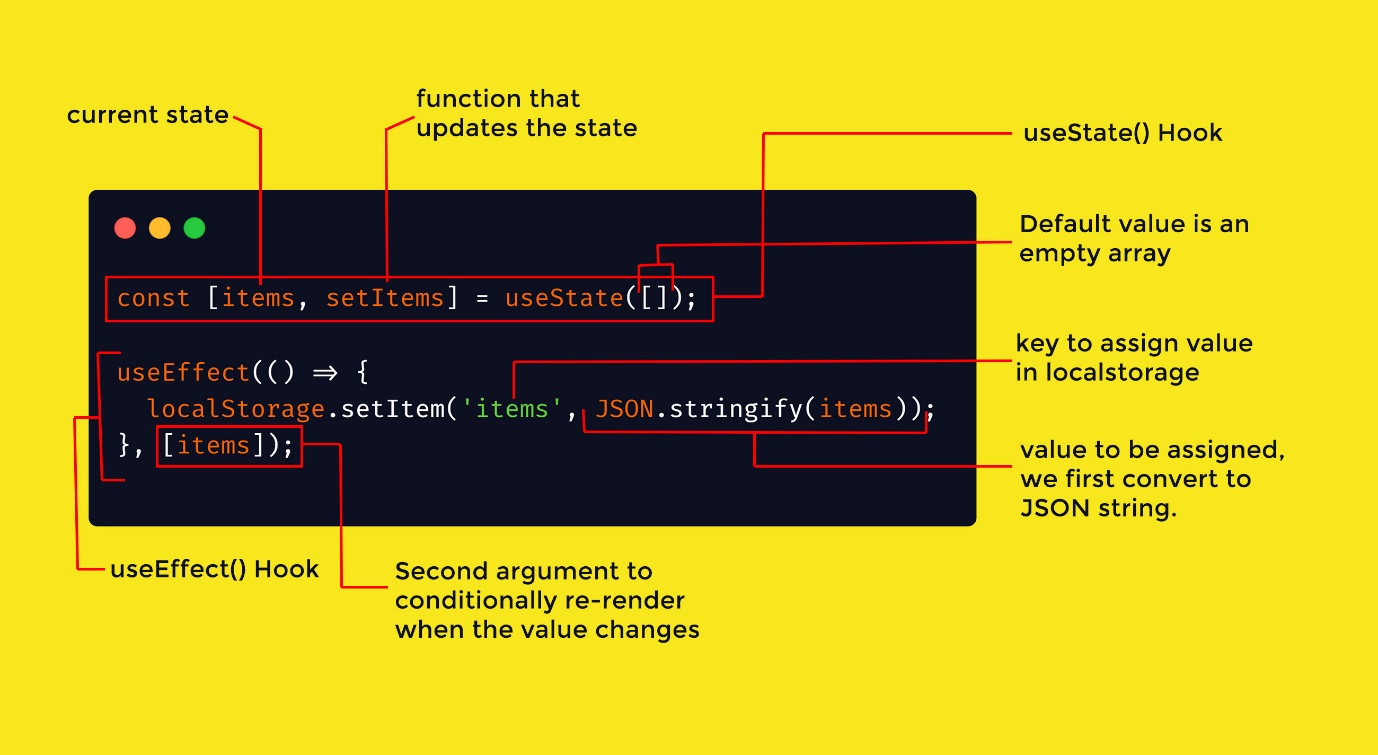
In the above code, we first created a state and assigned it an empty array (yours could be any other datatype). Second, we used useEffect() to add objects to localStorage whenever the value of our state changed. We did this by passing the state as the second argument.

Basically, this is the major code responsible for adding key-value pairs to localStorage:

localStorage.setItem('items', JSON.stringify(items));

Simply put, the preceding code names the key (lists) and then assigns a value to it, but we had to first ensure that the data we were adding was a JSON string.

We use JSON.stringify() to convert a JSON object to JSON text stored in a string, which can then be transmitted to the web server.

Structure of how hooks works with localstorage to set items

**How to Use the**getItem(**) Method**

This method retrieves objects from localStorage. There are other methods to accomplish this with React, but we will use the useEffect() hook because it is the best one.

The useEffect() hook helps us fetch all items on first render, which means that when the component mounts or re-renders, it obtains all of our data from localStorage.

Note that this is why we passed in an empty second argument.

const [items, setItems] = useState([]);

useEffect(() => {

  const items = JSON.parse(localStorage.getItem('items'));

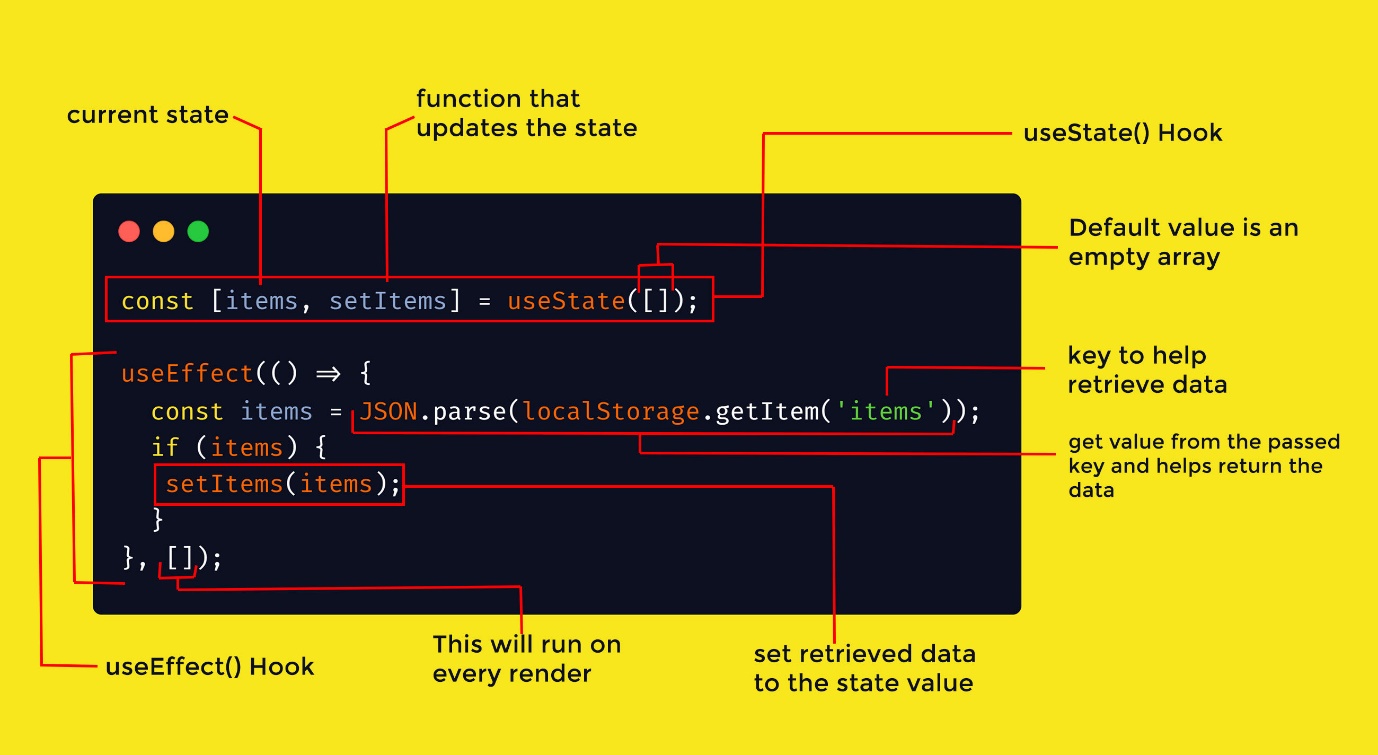
  if (items) {

   setItems(items);

  }

}, []);

It is important to remember that when we stored the data, we first converted it to a JSON string. This means that in order for us to now make use of it, we need to convert JSON string back to a JSON object. We do this with the JSON.parse() method.

Structure of how hooks works with localstorage to get items

**Conclusion**

In this article, we learnt how to use localStorage with React hooks, when to use it, and which hook to use.

**npm install --legacy-peer-deps** => to install all dependences of scripts in the package.json file.

**React Projects:**

<https://react-projects.netlify.app/>

freecodecamp.org