React is a popular JavaScript library for building user interfaces, especially single-page applications (SPAs). It was developed by Facebook and is maintained by Facebook and a community of developers.

React allows developers to create large web applications that can update and render efficiently in response to changing data.

**Key Concepts of React**

1. **Components:**
   * React is all about components. A component is a reusable piece of code that represents a part of the user interface. Components can be functional (stateless) or class-based (stateful).
   * Example of a functional component:

jsx

function Welcome(props) {

return <h1>Hello, {props.name}</h1>;

}

1. **JSX:**
   * JSX (JavaScript XML) is a syntax extension that allows you to write HTML elements in JavaScript. It makes it easier to create React elements.
   * Example:

jsx

const element = <h1>Hello, world!</h1>;

1. **Virtual DOM:**
   * React uses a virtual DOM to improve performance. The virtual DOM is a lightweight representation of the actual DOM. React updates the virtual DOM first, then compares it with the actual DOM and makes efficient updates to the actual DOM.
2. **State and Props:**
   * **State:** State is a built-in object in React that stores data about the component. It can change over time and re-render the component.
   * **Props:** Props (short for properties) are read-only attributes used to pass data from parent components to child components.
   * Example of using state in a class component:

jsx

class Clock extends React.Component {

constructor(props) {

super(props);

this.state = { date: new Date() };

}

componentDidMount() {

this.timerID = setInterval(() => this.tick(), 1000);

}

componentWillUnmount() {

clearInterval(this.timerID);

}

tick() {

this.setState({ date: new Date() });

}

render() {

return (

<div>

<h1>Hello, world!</h1>

<h2>It is {this.state.date.toLocaleTimeString()}.</h2>

</div>

);

}

}

1. **Lifecycle Methods:**
   * React class components have lifecycle methods that you can override to run code at specific times in a component's life. Common lifecycle methods include componentDidMount, componentDidUpdate, and componentWillUnmount.
2. **Hooks:**
   * Hooks are functions that let you use state and other React features in functional components. The most common hooks are useState and useEffect.
   * Example of using hooks in a functional component:

jsx

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

function Clock() {

const [date, setDate] = useState(new Date());

useEffect(() => {

const timerID = setInterval(() => tick(), 1000);

return () => clearInterval(timerID);

}, []);

function tick() {

setDate(new Date());

}

return (

<div>

<h1>Hello, world!</h1>

<h2>It is {date.toLocaleTimeString()}.</h2>

</div>

);

}

**Advantages of React**

1. **Component-Based Architecture:**
   * React's component-based architecture promotes reusability and modularity, making it easier to manage large applications.
2. **Unidirectional Data Flow:**
   * React follows a unidirectional data flow, which makes it easier to understand and debug applications.
3. **Virtual DOM:**
   * The virtual DOM improves performance by minimizing direct DOM manipulations and updating only the necessary parts of the UI.
4. **Strong Community and Ecosystem:**
   * React has a large and active community, providing plenty of resources, libraries, and tools to enhance development.

Choosing between Angular and React depends on the specific use cases and project requirements. Both are powerful frameworks/libraries for building web applications, but they have different strengths and best-fit scenarios.

**When to Prefer Angular**

1. **Large-Scale Enterprise Applications:**
   * Angular is well-suited for large-scale enterprise applications due to its comprehensive framework, which includes built-in features like dependency injection, form validation, routing, and state management. This makes it a great choice for complex projects with extensive requirements.
2. **Full-Fledged Framework:**
   * Angular is a full-fledged framework that provides a complete solution for building web applications. If you prefer a more opinionated approach where the framework provides a structured way to develop applications, Angular is a good fit.
3. **TypeScript Integration:**
   * Angular is built with TypeScript, offering strong typing and better tooling support. If your team prefers TypeScript or if you require strict type checking, Angular can be advantageous.
4. **Two-Way Data Binding:**
   * Angular's two-way data binding allows automatic synchronization of data between the model and the view, making it easier to handle form input and user interactions.
5. **Rich CLI and Tooling:**
   * Angular provides a powerful Command Line Interface (CLI) that simplifies project setup, development, testing, and deployment. It also offers built-in support for unit testing and end-to-end testing.

**When to Prefer React**

1. **Single-Page Applications (SPAs):**
   * React is ideal for building SPAs where the user interface dynamically updates without reloading the page. Its component-based architecture and virtual DOM make it efficient for creating interactive user experiences.
2. **Flexibility and Customization:**
   * React is a library, not a full-fledged framework, which provides more flexibility and freedom to choose additional libraries and tools. If you prefer a more flexible approach where you can pick and integrate different technologies, React is a good choice.
3. **Performance:**
   * React's virtual DOM efficiently updates the UI, leading to better performance, especially in applications with a lot of dynamic content and frequent updates.
4. **Learning Curve:**
   * React has a relatively smaller learning curve compared to Angular, making it easier for developers to get started. Its simplicity and ease of use are appealing for smaller projects or teams new to front-end development.
5. **Strong Community and Ecosystem:**
   * React has a large and active community, providing a wide range of third-party libraries, tools, and resources. This makes it easier to find solutions, tutorials, and support.

**Summary**

* **Choose Angular** for large-scale enterprise applications, if you prefer a full-fledged framework with built-in features, strong TypeScript integration, two-way data binding, and powerful CLI and tooling.
* **Choose React** for single-page applications, if you need flexibility and customization, better performance with the virtual DOM, a smaller learning curve, and access to a strong community and ecosystem.

**React Lifecycle**

The **React lifecycle** is an essential concept for any React developer to understand. Each component in React has a lifecycle that you can monitor and manipulate during its three main phases: **Mounting**, **Updating**, and **Unmounting**. These phases represent the stages a React component goes through from its creation to its removal from the DOM.

React components follow a predictable lifecycle, making it easier for developers to manage data fetching, subscriptions, DOM updates, and other side effects at appropriate times. In this article, we will explore the **three phases of the React lifecycle** and demonstrate how to use lifecycle methods effectively in class-based components.

**Table of Content**

* [React Lifecycle Phases](https://www.geeksforgeeks.org/reactjs-lifecycle-components/#react-lifecycle)
* [Phases of Lifecycle in React Components](https://www.geeksforgeeks.org/reactjs-lifecycle-components/#phases-of-lifecycle-in-react-components)
* [Implementing the Component Lifecycle methods](https://www.geeksforgeeks.org/reactjs-lifecycle-components/#implementing-the-component-lifecycle-methods)

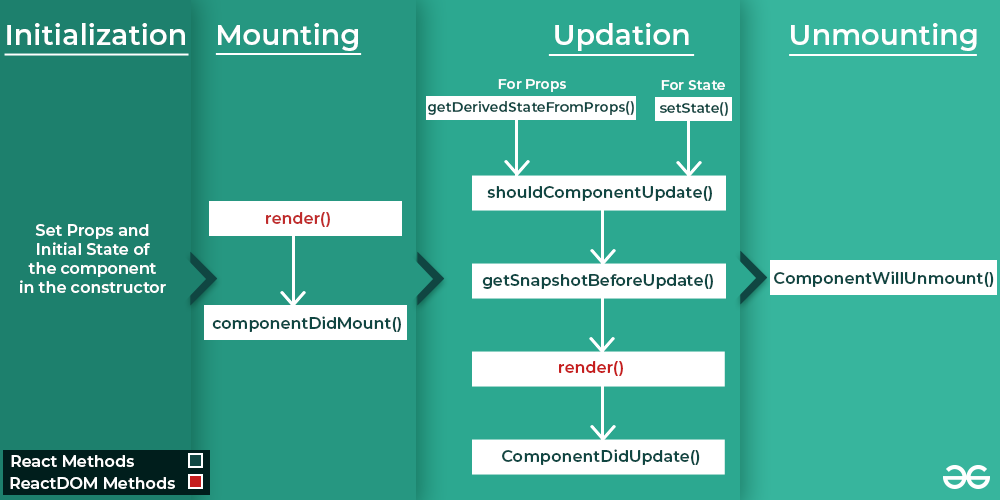
**React Lifecycle Phases**

**React Lifecycle** is defined as the series of methods that are invoked in different stages of the component’s existence. The definition is pretty simple but what do we mean by different stages? A[React](https://www.geeksforgeeks.org/react-tutorial/) Component can go through four stages of its life as follows.

The React lifecycle is divided into **three main phases**:

1. **Initialization phase:**This is the stage where the component is constructed with the given Props and default state. This is done in the constructor of a Component Class.
2. **Mounting Phase**: This phase begins when a component is created and inserted into the DOM.
3. **Updating Phase**: This occurs when a component is re-rendered due to changes in props or state.
4. **Unmounting Phase**: This is the final phase when a component is removed from the DOM.

React provides the developers with a set of predefined functions that if present are invoked around specific events in the lifetime of the component. Developers are supposed to override the functions with the desired logic to execute accordingly. We have illustrated the gist in the following diagram.



Now let us describe each phase and its corresponding functions.

**Phases of Lifecycle in React Components**

**Now lets Move to phases of React components**

**1. Initialization**

In this phase, the developer has to define the props and initial state of the component this is generally done in the constructor of the component. The following code snippet describes the initialization process.

class Clock extends React.Component {   
 constructor(props)   
 {   
 // Calling the constructor of   
 // Parent Class React.Component   
 super(props);   
   
 // Setting the initial state   
 this.state = { date : new Date() };   
 }   
}

**2. Mounting**

[Mounting](https://www.geeksforgeeks.org/explain-the-meaning-of-mounting-and-demounting/) is the phase of the component lifecycle when the initialization of the component is completed and the component is mounted on the [DOM](https://www.geeksforgeeks.org/dom-document-object-model/) and rendered for the first time on the webpage. Now React follows a default procedure in the Naming Conventions of these predefined functions where the functions containing “Will” represents before some specific phase and “Did” represents after the completion of that phase. The mounting phase consists of two such predefined functions as described below.

* constructor
* static getDerivedStateProps
* render()
* componentDidMount()

[constructor()](https://www.geeksforgeeks.org/react-js-constructor-method/)**:**

Method to initialize state and bind methods. Executed before the component is mounted.

constructor example:

*// Filename - src/index.js:*

**import** React **from** "react";

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

**class** Test **extends** React.Component {

**constructor**(props) {

**super**(props);

**this**.state = { hello: "World!" };

}

render() {

**return** (

<div>

<h1>

GeeksForGeeks.org, Hello

{**this**.state.hello}

</h1>

</div>

);

}

}

**const** root = ReactDOM.createRoot(

document.getElementById("root")

);

root.render(<Test />);

[static getDerivedStateFromProps](https://www.geeksforgeeks.org/react-js-static-getderivedstatefromprops/)

Used for updating the state based on props. Executed before every render.

Example:

*// Filename - src/index.js:*

**import** React **from** "react";

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

**class** Test **extends** React.Component {

**constructor**(props) {

**super**(props);

**this**.state = { hello: "World!" };

}

**static** getDerivedStateFromProps(props, state) {

**return** { hello: props.greet };

}

render() {

**return** (

<div>

<h1>

GeeksForGeeks.org, Hello

{**this**.state.hello}

</h1>

</div>

);

}

}

**const** root = ReactDOM.createRoot(

document.getElementById("root")

);

root.render(<Test greet="Geeks!"/>);

[render() method](https://www.geeksforgeeks.org/react-js-render-method/)**:**

Responsible for rendering JSX and updating the DOM.

**render() Example:**

*// Filename - src/index.js:*

**import** React **from** "react";

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

**class** Test **extends** React.Component {

render() {

**return** (

<div>

<h1>

GeeksforGeeks

</h1>

</div>

);

}

}

**const** root = ReactDOM.createRoot(

document.getElementById("root")

);

root.render(<Test />);

[**componentDidMount() Function**](https://www.geeksforgeeks.org/reactjs-componentdidmount-method/)

This function is invoked right after the component is mounted on the DOM i.e. this function gets invoked once after the render() function is executed for the first time

**componentDidMount() Example:**

*// Filename - src/index.js:*

**import** React **from** "react";

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

**class** Test **extends** React.Component {

**constructor**(props) {

**super**(props);

**this**.state = { hello: "World!" };

}

componentDidMount() {

**this**.setState({hello:"Geeks!"})

}

render() {

**return** (

<div>

<h1>

GeeksForGeeks.org, Hello

{**this**.state.hello}

</h1>

</div>

);

}

}

**const** root = ReactDOM.createRoot(

document.getElementById("root")

);

root.render(<Test />);

**3. Updation**

React is a JS library that helps create Active web pages easily. Now active web pages are specific pages that behave according to their user. For example, let’s take the GeeksforGeeks {IDE} webpage, the webpage acts differently with each user. User A might write some code in C in the Light Theme while another User may write Python code in the Dark Theme all at the same time. This dynamic behavior that partially depends upon the user itself makes the webpage an Active webpage.

Now how can this be related to Updation? Updation is the phase where the states and props of a component are updated followed by some user events such as clicking, pressing a key on the keyboard, etc. The following are the descriptions of functions that are invoked at different points of the Updation phase.

* getDerivedStateFromProps
* setState() Function
* shouldComponentUpdate()
* getSnapshotBeforeUpdate() Method
* componentDidUpdate()

[**getDerivedStateFromProps**](https://www.geeksforgeeks.org/react-js-static-getderivedstatefromprops/)**:**

getDerivedStateFromProps(props, state) is a static method that is called just before render() method in both mounting and updating phase in React. It takes updated props and the current state as arguments.

static getDerivedStateFromProps(props, state) {  
 if(props.name !== state.name){  
 //Change in props  
 return{  
 name: props.name  
 };  
 }  
 return null; // No change to state  
}

[**setState()**](https://www.geeksforgeeks.org/reactjs-setstate/)

This is not particularly a Lifecycle function and can be invoked explicitly at any instant. This function is used to update the state of a component. You may refer to [this article](https://www.geeksforgeeks.org/reactjs-state-react/) for detailed information.

this.setState((prevState, props) => ({  
 counter: prevState.count + props.diff  
}));

**setState Example:**

*// Filename - index.js*

**import** React **from** "react";

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

**class** App **extends** React.Component {

**constructor**(props) {

**super**(props);

**this**.state = {

count: 0,

};

}

increment = () => {

**this**.setState((prevState) => ({

count: prevState.count + 1,

}));

};

decrement = () => {

**this**.setState((prevState) => ({

count: prevState.count - 1,

}));

};

render() {

**return** (

<div>

<h1>

The current count is :{" "}

{**this**.state.count}

</h1>

<button onClick={**this**.increment}>

Increase

</button>

<button onClick={**this**.decrement}>

Decrease

</button>

</div>

);

}

}

**const** root = ReactDOM.createRoot(

document.getElementById("root")

);

root.render(

<React.StrictMode>

<App />

</React.StrictMode>

);

[**shouldComponentUpdate()**](https://www.geeksforgeeks.org/reactjs-shouldcomponentupdate-method/)

By default, every state or props update re-renders the page but this may not always be the desired outcome, sometimes it is desired that updating the page will not be repainted. The shouldComponentUpdate() Function fulfills the requirement by letting React know whether the component’s output will be affected by the update or not.

shouldComponentUpdate() is invoked before rendering an already mounted component when new props or states are being received. If returned false then the subsequent steps of rendering will not be carried out. This function can’t be used in the case of [forceUpdate().](https://www.geeksforgeeks.org/reactjs-forceupdate-method/) The Function takes the new Props and new State as the arguments and returns whether to re-render or not.

shouldComponentUpdate(nextProps, nextState)

It returns true or false, if false then render(), componentWillUpdate() and componentDidUpdate() method does not gets invoked.

[**getSnapshotBeforeUpdate() Method**](https://www.geeksforgeeks.org/reactjs-getsnapshotbeforeupdate-method/)

The getSnapshotBeforeUpdate() method is invoked just before the DOM is being rendered. It is used to store the previous values of the state after the DOM is updated.

getSnapshotBeforeUpdate(prevProps, prevState)

[**componentDidUpdate() Function**](https://www.geeksforgeeks.org/reactjs-componentdidupdate-method/)

Similarly this function is invoked after the component is rerendered i.e. this function gets invoked once after the render() function is executed after the updation of State or Props.

componentDidUpdate(prevProps, prevState, snapshot)

**4. Unmounting**

This is the final phase of the lifecycle of the component which is the phase of unmounting the component from the DOM. The following function is the sole member of this phase.

[**componentWillUnmount() Function**](https://www.geeksforgeeks.org/reactjs-componentwillunmount-method/)**:**

This function is invoked before the component is finally unmounted from the DOM i.e. this function gets invoked once before the component is removed from the page and this denotes the end of the lifecycle.

**componentWillUnmount() Example:**

**import** React **from** "react";

**class** ComponentOne **extends** React.Component {

*// Defining the componentWillUnmount method*

componentWillUnmount() {

alert("The component is going to be unmounted");

}

render() {

**return** <h1>Hello Geeks!</h1>;

}

}

**class** App **extends** React.Component {

state = { display: **true** };

**delete** = () => {

**this**.setState({ display: **false** });

};

render() {

**let** comp;

**if** (**this**.state.display) {

comp = <ComponentOne />;

}

**return** (

<div>

{comp}

<button onClick={**this**.**delete**}>

Delete the component

</button>

</div>

);

}

}

**export** **default** App;

We have so far discussed every predefined function there was in the lifecycle of the component, and we have also specified the order of execution of the function.

**Implementing the Component Lifecycle methods**

Let us now see one final example to finish the article while revising what’s discussed above.

First, create a react app and edit your **index.js** file from the src folder.

*// Filename - src/index.js:*

**import** React **from** "react";

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

**class** Test **extends** React.Component {

**constructor**(props) {

**super**(props);

**this**.state = { hello: "World!" };

}

componentDidMount() {

console.log("componentDidMount()");

}

changeState() {

**this**.setState({ hello: "Geek!" });

}

render() {

**return** (

<div>

<h1>

GeeksForGeeks.org, Hello

{**this**.state.hello}

</h1>

<h2>

<a

onClick={**this**.changeState.bind(

**this**

)}

>

Press Here!

</a>

</h2>

</div>

);

}

shouldComponentUpdate(nextProps, nextState) {

console.log("shouldComponentUpdate()");

**return** **true**;

}

componentDidUpdate() {

console.log("componentDidUpdate()");

}

}

**const** root = ReactDOM.createRoot(

document.getElementById("root")

);

root.render(<Test />);

**Output:**This output will be visible on the **http://localhost:3000**on the browser window.

**React Modules :**

In React, modules are used to organize and reuse code effectively. Modules help you keep your codebase modular and maintainable by allowing you to split your application into smaller, reusable pieces. Each module can export functionality (components, functions, constants, etc.) and import functionality from other modules.

**Exporting and Importing Modules in React**

1. **Exporting Modules:**
   * There are two ways to export modules in React: named exports and default exports.

**Named Exports:**

jsx

// File: utils.js

export const PI = 3.14;

export function add(a, b) {

return a + b;

}

**Default Export:**

jsx

// File: Header.js

export default function Header() {

return <h1>Welcome to My App</h1>;

}

1. **Importing Modules:**
   * You can import modules using the import statement.

**Importing Named Exports:**

jsx

// File: App.js

import { PI, add } from './utils';

console.log(PI); // 3.14

console.log(add(2, 3)); // 5

**Importing Default Exports:**

jsx

// File: App.js

import Header from './Header';

function App() {

return (

<div>

<Header />

<p>This is my app</p>

</div>

);

}

export default App;

**Example of Using Modules in a React Application**

Let's go through a full example of using modules in a React application to demonstrate how they work together.

1. **Create a Utility Module:** Save the following code in a file named utils.js:

jsx

// utils.js

export const PI = 3.14;

export function add(a, b) {

return a + b;

}

1. **Create a Header Component:** Save the following code in a file named Header.js:

jsx

// Header.js

import React from 'react';

export default function Header() {

return <h1>Welcome to My App</h1>;

}

1. **Create the Main App Component:** Save the following code in a file named App.js:

jsx

// App.js

import React from 'react';

import Header from './Header';

import { PI, add } from './utils';

function App() {

return (

<div>

<Header />

<p>This is my app.</p>

<p>PI: {PI}</p>

<p>2 + 3 = {add(2, 3)}</p>

</div>

);

}

export default App;

1. **Set Up the Entry Point:** Save the following code in a file named index.js:

jsx

// index.js

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App';

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

1. **Create the HTML File:** Create an index.html file in the public directory of your project with the following content:

html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>React App</title>

</head>

<body>

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

<script src="index.js"></script>

</body>

</html>

1. **Install Dependencies and Run the App:** Make sure you have react and react-dom installed in your project. You can create a new React app using create-react-app or set up your own project with the following dependencies:

sh

npm install react react-dom

1. **Start the Development Server:** If you used create-react-app, you can start the development server with:

sh

npm start

If you set up your own project, make sure you have a build tool like webpack configured to bundle your JavaScript files.

**React Js Services**

React JS has revolutionized the way we build web applications. It offers a component-based approach, making it easier to manage and maintain large applications. In this blog post, we’ll create a simple service in React and connect it to a view. This service will demonstrate how to manage state and pass data between components.

**Setting Up the Project**

Before we dive into coding, ensure that you have Node.js installed on your machine. Then, create a new React app by running the following command:

npx create-react-app react-service-example

cd react-service-example

**Creating the Service**

Let’s start by creating a simple service. In React, a service is typically a set of functions that handle data fetching, processing, or any other logic that isn’t directly related to rendering the UI.

1. **Create a New File for the Service:**  
   Inside your project, create a new file named SampleService.js in the src directory.
2. **Writing the Service Logic:**  
   In SampleService.js , we’ll create a function that simulates fetching data from an API.

const fetchData = async () => {

try {

// Simulate an API call

const response = await new Promise(resolve => {

setTimeout(() => resolve({ data: 'Sample Data' }), 1000);

});

return response.data;

} catch (error) {

console.error("Error fetching data:", error);

}

};

export { fetchData };

**Explore:**[**How Can You Pass Props to Children Components in React?**](https://www.crsinfosolutions.com/how-can-you-pass-props-to-children-components-in-react/)

**Creating the View**

Now, let’s create a view that uses this service.

1. **Modify App.js:**  
   Open App.js and make the following changes:

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

import './App.css';

import { fetchData } from './SampleService';

function App() {

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

useEffect(() => {

const loadData = async () => {

const result = await fetchData();

setData(result);

};

loadData();

}, []);

return (

<div className="App">

<header className="App-header">

<p>

Fetched Data: {data}

</p>

</header>

</div>

);

}

export default App;

In this code, we import the fetchData function from our service. We then use the useEffect hook to call this function when the component mounts. The fetched data is stored in the data state variable and displayed in the view.

**Explore:**[**Lifecycle Methods in React**](https://www.crsinfosolutions.com/understanding-lifecycle-methods-in-react-react-js-tutorial-5/)

**Running the Application**

To see your service and view in action, run the following command in the terminal:

npm start

This will start the development server and open your new React application in the browser.

CRS Info Solutions stands out for its exceptional [React.js training in Hyderabad](https://www.crsinfosolutions.com/react-js-training-hyderabad/), tailored specifically for students. Their program focuses on practical, hands-on learning, ensuring that students not only understand [React.js training Bangalore](https://www.crsinfosolutions.com/react-js-training-in-bangalore/) concepts but also apply them effectively in real-world scenarios. This approach has established CRS Info Solutions as a go-to destination for aspiring React.js developers in the region.

**React Routing :**

Routing in React is typically handled using a library called **React Router**. React Router is a powerful routing library that enables navigation among different components, changing the browser URL, and keeping the UI in sync with the URL.

**Key Concepts of React Router**

1. **Router:**
   * The Router component is a wrapper that enables the routing functionality. There are different types of routers like BrowserRouter and HashRouter.
   * BrowserRouter uses the HTML5 history API to keep the UI in sync with the URL.
   * HashRouter uses the hash portion of the URL (the part after #).
2. **Route:**
   * The Route component defines the mapping between a URL path and a component. When the URL matches the path, the corresponding component is rendered.
   * The Route component can also handle route parameters.
3. **Link:**
   * The Link component is used to create navigation links that allow users to navigate between different routes without triggering a full page reload.
4. **Switch:**
   * The Switch component is used to render only the first Route or Redirect that matches the current location.

**Example: Basic Routing in React**

Let's create a simple React application with routing.

1. **Install React Router:** First, make sure to install the React Router library in your project.

sh

npm install react-router-dom

1. **Create Home Component:** Save the following code in a file named Home.js:

jsx

// Home.js

import React from 'react';

function Home() {

return (

<div>

<h2>Home Page</h2>

<p>Welcome to the Home page!</p>

</div>

);

}

export default Home;

1. **Create About Component:** Save the following code in a file named About.js:

jsx

// About.js

import React from 'react';

function About() {

return (

<div>

<h2>About Page</h2>

<p>Welcome to the About page!</p>

</div>

);

}

export default About;

1. **Create Contact Component:** Save the following code in a file named Contact.js:

jsx

// Contact.js

import React from 'react';

function Contact() {

return (

<div>

<h2>Contact Page</h2>

<p>Welcome to the Contact page!</p>

</div>

);

}

export default Contact;

1. **Set Up Routing in App Component:** Save the following code in a file named App.js:

jsx

// App.js

import React from 'react';

import { BrowserRouter as Router, Route, Switch, Link } from 'react-router-dom';

import Home from './Home';

import About from './About';

import Contact from './Contact';

function App() {

return (

<Router>

<div>

<nav>

<ul>

<li><Link to="/">Home</Link></li>

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

<li><Link to="/contact">Contact</Link></li>

</ul>

</nav>

<Switch>

<Route path="/" exact component={Home} />

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

<Route path="/contact" component={Contact} />

</Switch>

</div>

</Router>

);

}

export default App;

1. **Set Up the Entry Point:** Save the following code in a file named index.js:

jsx

// index.js

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App';

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

1. **Create the HTML File:** Create an index.html file in the public directory of your project with the following content:

html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>React Router Example</title>

</head>

<body>

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

<script src="index.js"></script>

</body>

</html>

1. **Install Dependencies and Run the App:** Make sure you have react and react-dom installed in your project. You can create a new React app using create-react-app or set up your own project with the following dependencies:

sh

npm install react react-dom react-router-dom

1. **Start the Development Server:** If you used create-react-app, you can start the development server with:

sh

npm start

If you set up your own project, make sure you have a build tool like webpack configured to bundle your JavaScript files.

**Explanation of App.js**

* **Router Setup:**

jsx

import { BrowserRouter as Router, Route, Switch, Link } from 'react-router-dom';

* + Import the necessary components from react-router-dom.
* **Navigation Links:**

jsx

<nav>

<ul>

<li><Link to="/">Home</Link></li>

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

<li><Link to="/contact">Contact</Link></li>

</ul>

</nav>

* + Use the Link component to create navigation links. Clicking these links will navigate to the corresponding routes without reloading the page.
* **Routes:**

jsx

<Switch>

<Route path="/" exact component={Home} />

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

<Route path="/contact" component={Contact} />

</Switch>

* + Define the routes using the Route component. The path prop specifies the URL path, and the component prop specifies the component to render when the path matches.
  + The exact prop ensures that the Home component is only rendered when the path is exactly /.

**Functional components**

Creating functional components in React is a straightforward and efficient way to build UI components. They are simply JavaScript functions that return JSX. Here's a quick guide to get you started:

1. **Set Up Your Environment**: Make sure you have Node.js and npm installed. Create a new React project using create-react-app.
2. **Create a Functional Component**: A functional component is a JavaScript function that returns JSX. Here's a basic example:

jsx

import React from 'react';

function MyComponent() {

return (

<div>

<h1>Hello, World!</h1>

</div>

);

}

export default MyComponent;

1. **Using Props**: Functional components can accept props to make them more dynamic.

jsx

import React from 'react';

function Greeting(props) {

return (

<div>

<h1>Hello, {props.name}!</h1>

</div>

);

}

export default Greeting;

1. **Adding State with Hooks**: Use the useState hook to add state to functional components.

jsx

import React, { useState } from 'react';

function Counter() {

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

return (

<div>

<p>You clicked {count} times</p>

<button onClick={() => setCount(count + 1)}>

Click me

</button>

</div>

);

}

export default Counter;

1. **Using Effects with Hooks**: The useEffect hook lets you perform side effects in your functional components.

jsx

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

function Timer() {

const [time, setTime] = useState(new Date().toLocaleTimeString());

useEffect(() => {

const timerID = setInterval(() => {

setTime(new Date().toLocaleTimeString());

}, 1000);

return () => clearInterval(timerID); // Clean up the interval on unmount

}, []);

return (

<div>

<p>Current time: {time}</p>

</div>

);

}

export default Timer;

**Class components**

Class components in React are JavaScript ES6 classes that extend from React.Component and can have state and lifecycle methods. While functional components are preferred for most use cases with React hooks, class components are still important to understand. Here's a basic guide:

1. **Creating a Class Component**: A class component must include a render method that returns JSX.

jsx

import React, { Component } from 'react';

class MyComponent extends Component {

render() {

return (

<div>

<h1>Hello, World!</h1>

</div>

);

}

}

export default MyComponent;

1. **Using Props**: Class components can access props through this.props.

jsx

import React, { Component } from 'react';

class Greeting extends Component {

render() {

return (

<div>

<h1>Hello, {this.props.name}!</h1>

</div>

);

}

}

export default Greeting;

1. **Adding State**: State can be added by initializing this.state in the constructor and updating it using this.setState.

jsx

import React, { Component } from 'react';

class Counter extends Component {

constructor(props) {

super(props);

this.state = { count: 0 };

}

increment = () => {

this.setState({ count: this.state.count + 1 });

}

render() {

return (

<div>

<p>You clicked {this.state.count} times</p>

<button onClick={this.increment}>

Click me

</button>

</div>

);

}

}

export default Counter;

1. **Lifecycle Methods**: Class components can use various lifecycle methods, like componentDidMount, componentDidUpdate, and componentWillUnmount.

jsx

import React, { Component } from 'react';

class Timer extends Component {

constructor(props) {

super(props);

this.state = { time: new Date().toLocaleTimeString() };

}

componentDidMount() {

this.timerID = setInterval(() => this.tick(), 1000);

}

componentDidUpdate() {

// This method is called every time the component updates.

}

componentWillUnmount() {

clearInterval(this.timerID);

}

tick() {

this.setState({ time: new Date().toLocaleTimeString() });

}

render() {

return (

<div>

<p>Current time: {this.state.time}</p>

</div>

);

}

}

export default Timer;

**Passing data between components**

Passing data between components using props is a fundamental concept in React. Props, short for properties, allow you to pass data from a parent component to a child component. Here's how you can do it with an example:

**Example: Passing Data Between Components**

Let's create a simple application with a parent component (Parent) and a child component (Child).

1. **Parent Component**: The parent component will hold the state and pass it to the child component using props.

jsx

import React, { Component } from 'react';

import Child from './Child';

class Parent extends Component {

constructor(props) {

super(props);

this.state = {

message: 'Hello from Parent Component!'

};

}

render() {

return (

<div>

<h1>Parent Component</h1>

<Child message={this.state.message} />

</div>

);

}

}

export default Parent;

1. **Child Component**: The child component receives the data passed from the parent component via props.

jsx

import React from 'react';

function Child(props) {

return (

<div>

<h2>Child Component</h2>

<p>{props.message}</p>

</div>

);

}

export default Child;

**Explanation:**

* **Parent Component**:
  + The Parent component holds the state message.
  + The message state is passed to the Child component through the message prop: <Child message={this.state.message} />.
* **Child Component**:
  + The Child component receives the message prop and displays it: {props.message}.

**Passing Data Back to Parent Component**

To pass data from a child component back to the parent, you can pass a function as a prop. Here's how:

1. **Parent Component**:

jsx

import React, { Component } from 'react';

import Child from './Child';

class Parent extends Component {

constructor(props) {

super(props);

this.state = {

message: ''

};

}

handleChildData = (childData) => {

this.setState({ message: childData });

}

render() {

return (

<div>

<h1>Parent Component</h1>

<Child sendDataToParent={this.handleChildData} />

<p>Data from Child: {this.state.message}</p>

</div>

);

}

}

export default Parent;

1. **Child Component**:

jsx

import React from 'react';

function Child(props) {

const sendData = () => {

props.sendDataToParent('Hello from Child Component!');

}

return (

<div>

<h2>Child Component</h2>

<button onClick={sendData}>Send Data to Parent</button>

</div>

);

}

export default Child;

**Explanation:**

* **Parent Component**:
  + The Parent component passes the handleChildData method as a prop to the Child component: <Child sendDataToParent={this.handleChildData} />.
  + The handleChildData method updates the state with data received from the child component.
* **Child Component**:
  + The Child component calls the function passed from the parent component, sending data back to the parent: props.sendDataToParent('Hello from Child Component!').

**Events in React**

**Handling events in React is quite similar to handling events in regular DOM elements, but with some syntax differences. Here are the basics:**

**1. Handling Events in Functional Components**

**In a functional component, you can handle events using standard JavaScript event handlers. Here's an example:**

**jsx**

**import React from 'react';**

**function ButtonClick() {**

**const handleClick = () => {**

**alert('Button clicked!');**

**};**

**return (**

**<button onClick={handleClick}>**

**Click me**

**</button>**

**);**

**}**

**export default ButtonClick;**

**2. Handling Events in Class Components**

**In a class component, event handlers are typically methods on the class. You have to use this to refer to these methods. Make sure to bind the methods in the constructor or use arrow functions to avoid issues with this.**

**jsx**

**import React, { Component } from 'react';**

**class ButtonClick extends Component {**

**constructor(props) {**

**super(props);**

**this.handleClick = this.handleClick.bind(this);**

**}**

**handleClick() {**

**alert('Button clicked!');**

**}**

**render() {**

**return (**

**<button onClick={this.handleClick}>**

**Click me**

**</button>**

**);**

**}**

**}**

**export default ButtonClick;**

**3. Passing Arguments to Event Handlers**

**You can pass arguments to event handlers using arrow functions or the bind method. Here's an example of both approaches:**

**Using Arrow Functions:**

**jsx**

**import React from 'react';**

**function ButtonClick() {**

**const handleClick = (message) => {**

**alert(message);**

**};**

**return (**

**<button onClick={() => handleClick('Button clicked!')}>**

**Click me**

**</button>**

**);**

**}**

**export default ButtonClick;**

**Using bind Method:**

**jsx**

**import React, { Component } from 'react';**

**class ButtonClick extends Component {**

**constructor(props) {**

**super(props);**

**this.handleClick = this.handleClick.bind(this);**

**}**

**handleClick(message) {**

**alert(message);**

**}**

**render() {**

**return (**

**<button onClick={this.handleClick.bind(this, 'Button clicked!')}>**

**Click me**

**</button>**

**);**

**}**

**}**

**export default ButtonClick;**

**4. Synthetic Events**

**React events are called Synthetic Events. They are a cross-browser wrapper around the browser's native event. They have the same interface as the browser's native events, including stopPropagation() and preventDefault(), except they work identically across all browsers.**

**5. Preventing Default Behavior**

**You can prevent the default behavior of events using the preventDefault method.**

**jsx**

**import React from 'react';**

**function LinkClick() {**

**const handleClick = (event) => {**

**event.preventDefault();**

**alert('Link clicked!');**

**};**

**return (**

**<a href="https://www.example.com" onClick={handleClick}>**

**Click me**

**</a>**

**);**

**}**

**export default LinkClick;**