React Render HTML

Cmd create application:

Syntax:-

//npx create-react-app name of app

npx create-react-app sample-app

what is Node-Js …Runtime environment

cmd to install Node Modules // npm i

3 Types of Node Modules

Class Modules,Functional Modules,Form Modules(3rd party dependencies)

Node Modules : 3rd party libraries and dependencies

Public: index.html view of web page(Home Page)

Robots.txt: seo and avoid overloading

JSON(Javsacript Object Notation) it will be served as server also(incase no api)

Communicate data between applications of project

Src :- source folder . Whatever code we write that stored in Src

Package.json / package lock json

:package.json focus on meta data and desired dependencies

Package.lock json … determines the installation of exact versions of dependencies

READme first item when visiting repository

Gitignore: some of the files cant be tracked by Git

In Public Folder- index.html

SPA :- React used to build Ui components of SPA

Single Page Application??

Only one HTML file …..

Node Modules

Public: index.html -view of web

Src: Source

App.css styling of App Component

App.js :- functionality of App component

Index.js – applicable to all components of project(Functionality)

Index.css- applicable to all components of project(css part)

First:- Name of Project

ReadMe,gitIgnore :-Git…

Install Extension in VS Code

ES7 React/Redux/GraphQL/React-Native snippets

Playcode.io/React

Render HTML in React

```
import React from 'react';
import ReactDOM from 'react-dom/client';
const container=document.getElementById('root');
const root=ReactDOM.createRoot(container);
root.render(<p> Render HTML using React</p>);
```

createRoot(); Function …invoke and take the HTML elements

render();method… whatever HTML we want to render …it will be rendered

Create one variable and display HTML in Node

```
import React from 'react';
import ReactDOM from 'react-dom/client';
const myelement=
(
  <table>
    <tr>
       <th> Name </th>
    </tr>
      <td> John </td>
   </tr>
   </tr>
     <td> Elsa</td>
   </tr>
  </table>
);
const container=document.getElementById('root');
const root=ReactDOM.createRoot(container);
root.render(myelement);
```

s

?? What is the out put

ES6 :-2015

Classes (similar to the function)

Class car{

Constructor(name)

this.model="name";

}

Arrow Functioins

Variables (let ,const)

Destructuring

Ternary Operator (conditional if-else)

Spread Operator …

Modules import & export

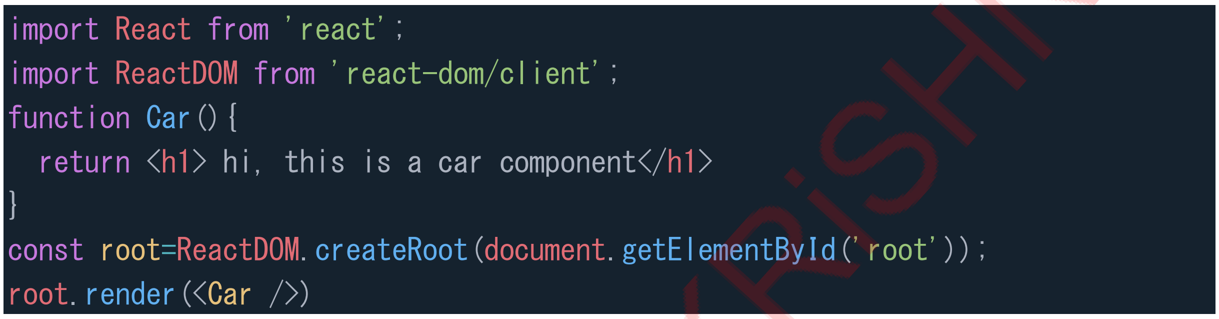
Map method (to make new array)

Components:-

Class (render())and Functional(js functions) components

MUST :-naming starts with capital letters

Functional component Example of component Car



```
import React from 'react';
import ReactDOM from 'react-dom/client';
function Car(){
  return <h1> hi, this is a car component</h1>
}
const root=ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car />)
```

functional component is just like javascript pure function accepts props as an argument and returns React element(JSX). Class component is extension from react

create a render function that returns React element

there is no render method used in functional components

differ in syntax and lifecycle

functional components use Hooks , Class use react life Cycle methods

class will hav state and lifecycle methods

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>;

  }

}
```

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

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

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

Props are arguments passed into React components.

Props are passed to components via HTML attributes.

`props` stands for properties.

React Props

React Props are like function arguments in JavaScript *and* attributes in HTML.

To send props into a component, use the same syntax as HTML attributes:

Example

Add a "brand" attribute to the Car element:

```
const myElement = <Car brand="Ford" />;
```

The component receives the argument as a `props` object:

Example

Use the brand attribute in the component:

```
function Car(props) {

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

}
```

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

function Car(props) {
  return <h2>I am a { props.brand }!</h2>;
}

const myElement = <Car brand="Ford" />;

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

Example

Send the "brand" property from the Garage component to the Car component:

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

function Car(props) {
  return <h2>I am a { props.brand }!</h2>;
}

function Garage() {
  return (
    <>
          <h1>Who lives in my garage?</h1>
          <Car brand="Ford" />
    </>
  );
}

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

Events in React

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()"`.

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

```
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.

```
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.

```
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 />);
```

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.

```
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} />);
```

```
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.

```
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} />);
```

```
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

```
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} />);
```

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.

```
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.
*/
```

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.

```
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>
    </>
  );
}

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

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:

```
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.

```
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>`:

```
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.

```
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.
*/
```

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:

```
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:

```
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 />);
```

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.

```
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: Get your own React.js Server

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;
```

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

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

```
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 />);
```

Solution

To fix this, we can use `memo`.

Use `memo`to 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 />);
```

There are three main ways that components communicate in React:

- Props:

  Props are the most common way for components to communicate. They are immutable data that is passed from a parent component to a child component. Props can be any type of data, including objects, arrays, strings, numbers, and functions.

- State:

  State is data that is owned by a component itself. It can be changed over time by the component, and the component will re-render whenever its state changes. State can only be accessed by the component that owns it.

- Events:

  Events are a way for components to communicate with each other asynchronously. When an event is triggered, all components that have subscribed to that event will be notified. Events can be used to implement things like buttons, forms, and other interactive elements.

Here are some examples of how components can communicate with each other using props, state, and events:

- Props:

  A parent component can pass props to a child component to provide the child component with data that it needs to render. For example, a parent component might pass the child component with a list of items to render.

- State:

  A child component can update its state in response to user input or other events. When the child component's state changes, the component will re-render. The parent component can then access the child component's state to update its own rendering.

- Events:

  A component can emit an event when something happens, such as when a button is clicked. Other components can then subscribe to the event and listen for it. When the event is triggered, the subscribed components will be notified.

Reading Time: 6 minutes

In modern front-end frameworks/libraries, the entire page is divided into multiple smaller components. Component-based architecture makes it easy to develop and maintain an application. During the development, you may come across a situation where you need to share the data with other components. In this blog, we're going to learn about the possible ways of achieving communication between React components.

React provides several methods for handling the component communication, each with its appropriate use cases.

-
    - **Parent / Child communication**
        - **Parent to child communication**
        - **Child to parent communication**
    - **Context API**
    - **Centralized state with Redux**
    - **Event bus**

• Parent / Child communication

○ Parent to child communication:

In React, parent components can communicate to child components using a special property defined by React called Props. *Props are read-only and they're passed from parent to child component via HTML attributes.*

Example:

Let's say that we have a parent component (*UserList*) that wants to display the user list and it passes the individual user information to the child component (*UserDetails*) to render the user info.

data.js

```
export const userData = [
      {
        id: 1,
        name: "Leanne",
        city: "McKenziehaven",
      },
      {
        id: 2,
        name: "Howell",
        city: "McKenziehaven",
      },
      {
        id: 3,
        name: "Bauch",
        city: "McKenziehaven",
      },
      {
        id: 4,
        name: "Patricia",
        city: "McKenziehaven",
      },
    ]
```

index.js

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

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

App.js

```
import "./App.css";
import React from "react";
import { UserList } from "./UserList";
import { userData } from "./data";

function App() {
  return <UserList userData={userData} />;
}

export default App;
```

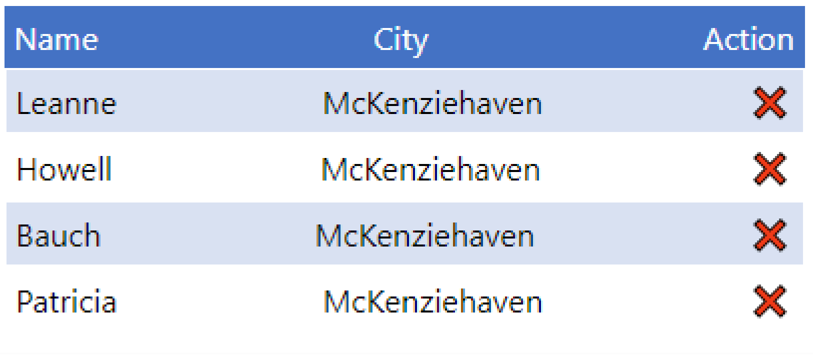
UserList.js

```
import React, { useState } from "react";
import { UserDetails } from "./UserDetails";

export const UserList = ({ userData = [] }) => {
  const [users, setUsers] = useState(userData);

  return (
      <div className="user-list">
        <div className="header">
          <span>Name</span>
          <span>City</span>
          <span>Action</span>
        </div>
        {users.map(({ id, name, city }) => (
          <UserDetails
            key={id}
            id={id}
            name={name}
            city={city}
          />
        ))}
      </div>
  );
};
```

UserDetails.js

```
import React from "react";

export const UserDetails = (props) => {
  const { id, name, city } = props;

  return (
    <div className="child-container user">
      <span>{name}</span>
      <span>{city}</span>
      <span className="action" title="Delete">
        ✕
      </span>
    </div>
  );
};
```

<u>**Output:**</u>



Name	City	Action
Leanne	McKenziehaven	❌
Howell	McKenziehaven	❌
Bauch	McKenziehaven	❌
Patricia	McKenziehaven	❌

-
    - ## Child to parent communication:

      Data from a child component to parent component can be passed using a callback. We can achieve this by following the below steps.

        - Create a callback method in the parent component and pass it to the child using props.
        - Child component can call this method using "this.props.[callbackFunctionName]" from the child and pass data as an argument.

Example:

Let´s say that we want to delete a user from the list and as the child component is responsible for displaying the user info, the delete button is rendered by the child component but the child component doesn't have the complete user list to delete a specific user. So, to achieve it we ́ll pass a callback function from the parent component and the child component will call that function when we click on the delete button.

Let's see it with the code.

Create a function for deleting an user in the *"UserList"* component.

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

const deleteUser = useCallback(
    (userId) => {
      setUsers(users.filter(({ id }) => id !== userId));
    },
    [users]
  );
```

Pass the function as a prop to the "UserDetails" component.

```
<User key={id} id={id} name={name} city={city} clickHandler={deleteUser} />
```

In the "*UserDetails*" component attach that function to the onClick event of the delete icon.

```
const { id, name, city, clickHandler } = props;
  return (
    <div className="child-container user">
      <span>{name}</span>
      <span>{city}</span>
      <span className="action" title="Delete" onClick={() =>
clickHandler(id)}>
          ✕
      </span>
    </div>
  );
```

- ## Context API

React Context API is introduced with React v.16.3. It allows us to pass data through our component trees, giving our components the ability to communicate and share data at different levels.

Let's explore how we can use React Context.

Create ThemeContext.js in src/ folder .

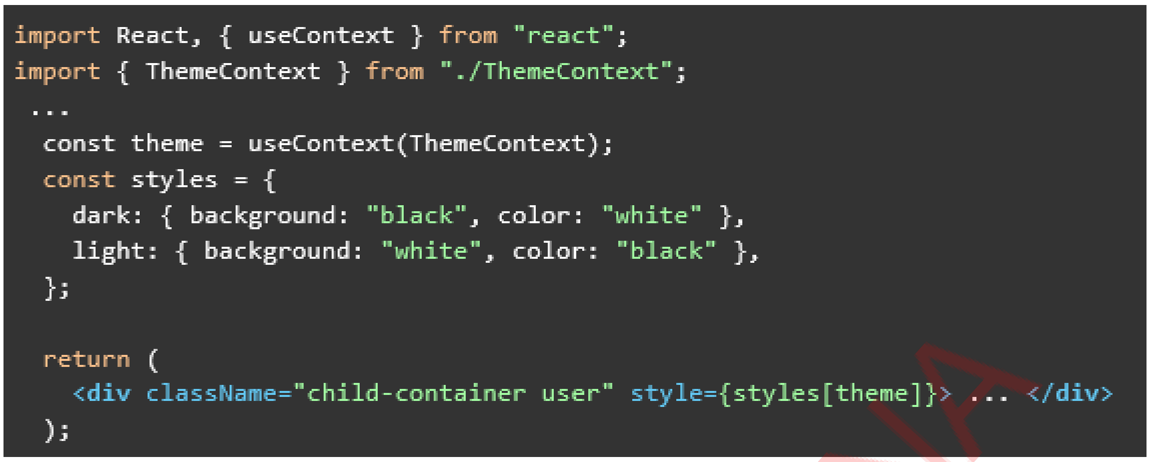
ThemeContext.js

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

export const ThemeContext = createContext("dark");
```

In the "*UserList*" component import the ThemeContext Wrap the child component with "ThemeContext.Provider"

```
import { ThemeContext } from "./ThemeContext";

return (
    <ThemeContext.Provider value="dark">
      <div className="user-list">...</div>
    </ThemeContext.Provider>
  );
```

In the "*UserDetails*" component with the help of useContext get the theme and apply the styles based on the theme.



```
import React, { useContext } from "react";
import { ThemeContext } from "./ThemeContext";
 ...
  const theme = useContext(ThemeContext);
  const styles = {
    dark: { background: "black", color: "white" },
    light: { background: "white", color: "black" },
  };

  return (
    <div className="child-container user" style={styles[theme]}> ... </div>
  );
```

If you decide to use React Context API then you should be aware of its potential performance issue. You might add to too many components where they aren't needed. To prevent re-rendering, be sure to place contexts only in the components that require them.

When to use:

Context is primarily used when some data needs to be accessible by many components at different nesting levels.

Routing in reactJS is the mechanism by which we navigate through a website or web-application. Routing can be server-side routing or client-side routing. However, React Router is a part of client-side routing. Before proceeding further it is highly recommended you go over the official **React.js certification course** to familiarize yourself with the fundamentals of React.

React Routing without any knowledge of it can be manually implemented using useState and JSX for conditioning. But being inefficient for large-scale applications like e-commerce, it still can act as a boilerplate for understanding routing and act as a base for React JS router for example.

```
const[page, setpage] = useState("products")

const routeTo = (newpage) =>{
  setpage(newpage)

}

<button onClick={()= routeTo("cart")}> Cart />

{page === "cart" && (

<Cart cart={cart} setcart={setcart}/>)}
```

Code

Server-side routing results in every request being a full page refresh loading unnecessary and repeated data every time causing a bad UX and delay in loading.

With client-side routing, it could be the rendering of a new component where routing is handled internally by JavaScript. The component is rendered only without the server being reloaded.

React Router helps us to navigate the components and build a single-page application without page-refresh when the user navigates making it an effective UX experience.

Web-Development is the heart of routing. Routing takes place efficiently in websites and so to understand it, one must know a basic boilerplate of **Web Development complete course** followed by a React.js certification course to rocket-start React-Routing!

Basic Routing

React Router API

Browser Router or React Router API is a most popular library for routing in react to navigate among different components in a React Application keeping the UI in alignment with URL.

According to react-router official docs, " React Router is a fully-featured client and server-side routing library for React, a JavaScript library for building user interfaces."

Various Packages in React Router Library

There are 3 different packages for React Routing.

1. **react-router:** It is the heart of react-router and provides core functionalities for routing in web applications and all the packages for react-router- dom and react-router-native
2. **react-router-native:** It is used for routing in mobile applications
3. **react-router-dom:** It is used to implement dynamic routing in web applications.

How Routing Works / Routing Mechanism

We know routing is user-side navigation to different pages or different pages on a website. Every rendering mechanism has its own routing way to navigate but dealing with React Router here that comes under Client Side Routing but eventually under Client Side Rendering has a dynamic mechanism to look upon.

Client Side rendering orders the server to deal with data only whereas the rest of all rendering and routing is handled by the client itself.

Traditional routing has always requested the server to provide different index.html of different pages, but Client Side Rendering will only return one index.html file for Client Side Routing.

This will give you a smooth Single Page Application Routing experience with forward and backward route ability using history API keeping the URL updated as well.

Installation and the Steps for React Router

Prerequisite for react-router dom install:

1. You must have a react app created using create-react-app
2. The react app must be running with dummy code to proceed with creating react app router

React Router will help us make a dynamic navbar with different links to route on, resembling a blog application where every link routes us to a different new page.

Step 1: Run the following commands in terminal

```
npm install react-router-dom@6

or

yarn add react-router-dom@6
```

Code

Step 2: The package installs after the completion of npm and a message is received on the terminal varying with your system architecture.

...

```
+ react-router-dom@6

added 11 packages
from 6 contributors and audited 1981 packages in 24.897s


114 packages are looking for funding

  run `npm fund` for details


found 0 vulnerabilities
```

Code

And that’s it, we are ready to route!

Now let's come to the source code after the fundamental installation:

Source Code and Snippets

Step 1: After the installation of react-router-dom we can ensure that the package is successfully installed or not by checking the package.json file to see the installed react-router-dom module and its version.

```
{
   "name" : "reactApp",

   "version" : "1.0.0"

   "description" : "It is.a react app"

  "dependencies" : {

        "react" : "^17.0.2",

       "react-dom" : "^17.0.2",

       "react-icons" : "^4.3.1",

       "react-router" : "^6.2.1",

       "react-router-dom" : "^6.2.1"

},
```

Code

Step 2: Then you go straight to your index.js main page to activate BrowserRouter throughout the application running in the App.js file.

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

import ReactDOM from "react-router-dom"

import App from "./App"

import {BrowserRouter} from "react-router-dom"

const rootElement = document.getElementById("root")

ReactDOM.render(

<StrictMode>

    <BrowserRouter>

      <App />

   </BrowserRouter>
</StrictMode>,

  rootElement

);
```

Code

Step 3: Now, we can make directories for the components or the pages we want to render. Either you can make separate folders or can have one folder with all components. Using a terminal or with a new tab, folders can be created with ease.

```
mkdir src/components/Home

mkdir src/components/About
```

Code

Now we will create a component inside each directory we created above. Here we will first create a Home.js file for the Home directory.

```
nano src/components/Home/Home.js
```

Code

Then add the basic component rendering code for it.

```
function Home() {

    return (

       <div>

            <h1> This is the home page </h1>

      </div>

    );

}

export default Home;
```

Code

Followed by creating an About.js file for the About directory.

```
nano src/components/About/About.js
```

Code

Then add the basic component rendering code for it.

```
function About() {

    return (

       <div>

            <h1> This is the about page </h1>

      </div>

    );

}

export default About;
```

Code

Step 4: Now come to the main App.js file which is the core of implementing all we have defined and declared till now by defining routes for each component and where and which component they will render when the path matches with the base URL entered or clicked by the user.

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

import Home from "./components/Home/Home"

import About from "./components/About/About"

function App(){

   return (
      <div className="App">

        <Routes>

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

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

         </Routes>

    </div>

)}

export default App
```

Code

This is how we install and set up the basic boilerplate using React Router. After that, it can be extended with its components navigating with respect to website requirements.

React Router: Challenges and Debugging

React Router is simple to use if you follow and understand its basic template of it.

The challenges involved when you installed it on the terminal and tried to activate BrowserRouter but then also routing did not happen. Debugging comes with hands-on practice when you code the concept you visualize with the understanding of the concept learned here.

1. Have <Link> component inside <BrowserRouter> component because let's say if you have Header.js component with code :

```
import React from 'react';

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

const Header = () => {

    return (
        <div className="App">

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

             <Link to="/" >  HomePage </Link>

        </div>

    );

};
```

Code

and App.js with the rendering looks like :

```
import React from 'react';

import { BrowserRouter, Route, Link } from 'react-router-dom'

import Home from "./components/Home/Home"

import About from "./components/About/About"

import Header from './Header';

const App = () => {

  return (

    <div >

      <Header />

      <BrowserRouter>

        <div>

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

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

        </div>
      </BrowserRouter>

    </div>

  );

};

export default App;
```

Code

Here Header.js is using the <Link> component in the Header.js file but <Header> is placed outside <BrowserRouter> in app.js file making the error displayed: “component that is not the child of <Router> cannot contain its components as well.

2. Route is the child component of Routes, it must be used like taking Routes as parent component. Here the problem lies in the react-router version installation, react router 6 version does not allow Route to render without wrapping it up in Routes parent component just like:

```
function App() {

    return (

      <div>

      <Routes>

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

       <Route path="/shop" element={<Shop/>} />

      </Routes>

   </div>

  );

 }
```

Code

3. Do not use anchor tags instead of <Link> components because using anchor tags would not allow applications to remain Single Page Application (SPA). HTML anchor tag would trigger a page reload or page refresh when clicked.

4. Use the default Route always at the end while using switch components. Default Route is in the form of Redirect or Navigate in react router-dom@6 version.

Redirection happens when a login button is clicked on the Profile page redirecting you to <Login> component. Now <Redirect> is deprecated and {useNavigate} is currently in use with react-router latest version.

```
import React from "react"

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


export default function Profile() {

   let navigate = useNavigate()

   return (

   <div>

         <h2> This is profile </h2>

         <button> onClick ={()=>{ navigate("/about")}}> Login

         </button>

   </div>

);
```

Code

5. Routes are used rather than switches in react-router-dom@6 install

```
<BrowserRouter>
```

```
    <Routes>

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

      </Route>

    </Routes>

  </BrowserRouter>
```

Code

6. exact keyword must be used to match the component's route paths precisely. If we have code somewhere like this:

```
<BrowserRouter>

            <Switch>

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

                <Route path="/home" component={Main} />

            </Switch>

</BrowserRouter>
```

Code

The problem lies here with the Home Route which is the base route. React is needed to tell other routes are also appending with the "/" using exact.

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

Code

How to Use React Router in Typescript

React-Router@5 is generally used for routing generally, but React-Router@6 is great for typescript programmers shipping type definitions. With packages installed react-router with typescript quick starts normally.

1. npm install react-router-dom

2. npm install @types/react-router-dom

Typescript is an extension of JavaScript, adding one good layer of safety and typing into the existing code, typing makes it easy for programmers to trace the problem.

React Router Examples

React Router can be implemented in any case where the primary requirement is to navigate. Navbars, User Registration and Login can be implemented too.

The segregation of components into their pages is considered to be react routing best practices.

Step 1: Activate BrowserRouter in index.js file

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

import ReactDOM from "react-router-dom"

import App from "./App"

import {BrowserRouter as Router} from "react-router-dom"


const rootElement = document.getElementById("root")

ReactDOM.render(

<StrictMode>

    <BrowserRouter>

      <App />

   </BrowserRouter>

</StrictMode>,

  rootElement

);
```

Code

Step 2: Make different components to render on routing: Home, Wishlist, Cart, No Match.

```
import React from "react"

function Home() {

    return (

       <div>

            <h1> This is the home page </h1>

      </div>

    );

}

export default Home;


import React from "react"


export default function Cart() {

    return  <h1> This is Cart </h1>

}

import React from "react"

export default function WishList() {

 return  <h1> This is Wishlist </h1>

}
```

Code

```
import React from "react"


export default function NoMatch() {

    return  <h1> This is  404 </h1>

}
```

Code

Step 3: Route them in app.js file accordingly

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

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

import Home from "./pages/Home";

import Cart from "./pages/Cart";

import WishList from "./pages/WishList";

import NoMatch from "./pages/NoMatch";

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


export default function App() {

   return (

      <div className="App">

        <nav>

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

            <Link to ="/cart"> Cart </Link> ||

            <Link to ="/wishlist"> Wishlist </Link>
```

```
       </nav>

       <Routes>

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

          <Route path ="/cart" element= {<Cart />}/>

          <Route path ="/wishlist" element= {<WishList />}/>

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

       </Routes>

  </div>

);

}
```

Code

The react router navbar would look like this:

Different Types of Routers in React Router

React Routers gives us 3 types of routing in reactJS-

1. Browser Router

It is the most common type and most used type of router that uses HTML5 history API (pushState, replaceState, and the popstate event) making your UI in sync with the URL.

2. Hash Router

As the name sounds this type of router uses the hash portion of the URL keeping your UI in sync with the URL.

3. Memory Router

A router that keeps the history of your "URL" in memory does not like to put it in the address bar. Often used but useful in testing and non-browser environments like React Native.

Components in React Router and their Explanation

React Router DOM in reactJS is categorized into three primary components :

1. Routers

As we know, react-router-dom is an npm package that helps in dynamic routing, and provides <BrowserRouter> and <HashRouter>.

A) BrowserRouter: It is one of the parent components that is used to store all <Route> components that basically instruct the react app to navigate the components based on the route requested by the client. It is usually given an alias name 'Router' for ease of reference. It allows us to frame a complete and proper URL for navigation. Also, an additional point includes BrowserRouter gives us an inch of pain to handle the different routes configurations on the server-side This matters when you need to deploy large-scale production apps.

The syntax works like:

```
<BrowserRouter

       basename = {{optionalString}

       forceRefresh={optionalBool}

       getUserConfirmation={optionalFunc}

       keyLength = {optionaNumber}

>

  <App/>

</BrowserRouter>
```

Code

- basename: The basename prop is used to provide a base URL path for all the locations in the application.
- forceRefresh: It is a Boolean prop when set to true, refreshes and reloads the entire page with any route navigated across the same page.
- getUserConfirmation: A function that is used to confirm navigation but by default windows confirm it with window.confirm().
- keyLength: Location is a client-side library that allows us to implement routing, each location has the unique key but by default the key length: 6

The fundamental react router dom example is as follows:

```
import "./styles.css"

import {BrowserRouter} from "react-router-dom"

import {Routes, Route} from "./pages/Home"

import ProductDetail from "./pages/ProductDetail"

export default function App( ) {

return

   <BrowserRouter>

      <div className ="App"/>

     <Routes>

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

         <Route path = "/product-detail" element =
{<ProductDetail />} />

     </Routes>

  </div>

</BrowserRouter>

)
```

Code

HashRouter: It is the same as BrowserRouter but covers the two disadvantages of it when the BrowserRouter is not able to handle some old legacy servers or static servers like GitHub Pages, then HashRouter replaces it. As the name suggests, it first and foremost adds # to the forming base URL, it is used when there is no dependency on server-side configurations. It is just a # that is added between domain and path, and due to no dependency on history API, it works well in legacy servers. Also as it does not send anything written after # to the server request for say: https:// localhost:3000/#/about, the request will always go to / in the server, making the server happy to handle one single file only and the rest route path is handled at client side only to navigate the client to correct the route instantly.

2. Route Matchers

They are the matchers to navigate clients to and from the correct URL requested for using:

a. Route: It is the most basic component that maps location to different react components to render a specific UI when the path matches the current URL. It is an alternative to an if-statement logic for saying if want to go to the requested /about the path, the route is responsible for rendering that specific component matching across the current URL like <Route path ="/about" component ={ About }/>

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

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

<Route path ="/blogs" component={<Blogs />}/>
```

Code

b. Switch: This component works similarly to the switch statement. It cuts down the exhaustive checking of each route path with the current URL instead the switch once gets the correct route path it returns back from there instead of checking till the last route path is written.

```
import {BrowserRouter , Switch, Route } from "react-router-
dom"

<Switch>

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

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

     <Route path ="/blogs" component={<Blogs />}/>

</Switch>
```

Code

Route Changers: There are three types of navigators or route changes :

1. **Link:** It is one of the primary ways to navigate the routes instead of putting it in the address bar you render it on UI as fully accessible just like anchor <href> tags of HTML with the main difference of anchor tags reload the UI to navigate, but links do not reload it again
2. **NavLink:** It is an extended version of Link such that it allows us to see whether the < Link > is active or not. This component provides isActive property that can be used with the className attribute.

3. Redirect: It is a kind of force navigation that will redirect user to the specific page programmers want if such a route is yet not present in the application to navigate forward to. It is a substitute for 404 error pages depending on the website requirement. It exists to have a strong use-case when user signs up on any web-application he is automatically redirected to the login page making UI experience more efficient.

```
<Switch>

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

      <NavLink to ="about" className={({ isActive }) =>

(isActive ? 'active': 'inactive')}>About</NavLink>
```

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

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

</Switch>
```

Code

What is the Difference Between React Router and React Router DOM?

These two seem identical. react-router is the core npm package for routing, but react-router-dom is the superset of react-router providing additional components like BrowserRouter, NavLink and other components, it helps in routing for web applications.

Looking to master Python? Discover the **best Python course** that guarantees to take your skills to the next level. Unleash your coding potential and become a Python pro today!

Conclusion

Till now we learned the basics and fundamentals of react router and its routing mechanism to start and how it helps to render specific components on the go in web applications. The more you extend towards the application requirement the more specific use-case can be drawn from it. The crux of the reactJS router works same in react router class component as well , just the syntax differences can be seen.
Are you looking forward to learning to react from scratch? Check out the **React.js certification course from KnowledgeHut**, and pick the right one that best suits your interest.

Frequently Asked Questions(FAQs)

1. Which Router is best for React JS?

Generally, BrowserRouter works best for building learning projects and small scale web-applications. It is just suited less at the production level due to its unavailability in legacy servers ending it up with a Cannot GET Route error plus it needs server configuration for every route created at the client side. Hence extra configuration at the server side plus limitation to SPA(Single Page Application) exists.

2. Do we need a React Router with the next JS?

Generally, react-router-dom is an npm package installed for routing but nextJS has its own inbuilt router giving ease of routing with next / link using which navigation is implemented.

3. What is the difference between a router and a switch?

The switch works just as the switch statement, by going through each and every route but once it matches the current URL with the route path, it returns back from there rendering that route component specifically. But Router is a standard library under which the switch is a component that helps users to navigate via the implementation of routing in web applications.

4. How do I enable routing in React?

Routing in react is easily enabled while installing react-router-dom npm package and then you are good to go for ease of navigation .

Here is an example of how to implement navigation in React JS using React Router:

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

const Home = () => {
  return (
    <div>
      <h1>Home</h1>
      <Link to="/about">About</Link>
    </div>
  );
};

const About = () => {
  return (
    <div>
```

```
      <h1>About</h1>
      <Link to="/">Home</Link>
    </div>
  );
};

const App = () => {
  return (
    <Router>
      <Route exact path="/" component={Home} />
      <Route path="/about" component={About} />
    </Router>
  );
};

export default App;
```
Use code with caution.

Learn more

In this example, we have two components: **`Home`** and **`About`**. The **`Home`** component has a link to the **`About`** component, and the **`About`** component has a link to the **`Home`** component. We use the **`BrowserRouter`** component from React Router to wrap our application, and we use the **`Route`** component to define the different routes in our application.

When a user clicks on the link to the **`About`** component, the **`Router`** component will render the **`About`** component. When a user clicks on the link to the **`Home`** component, the **`Router`** component will render the **`Home`** component.

This is just a simple example of how to implement navigation in React JS using React Router. There are many other ways to implement navigation in React JS, and the best way to do it will depend on the specific needs of your application.

Here are the different phases of a React component lifecycle:

- Mounting: This is the first time a component is inserted into the DOM.
- Updating: This is when a component's state or props change.
- Unmounting: This is when a component is removed from the DOM.

There are also a few lifecycle methods that are called during each phase:

- Mounting:
  - **`constructor()`**
  - **`static getDerivedStateFromProps()`**
  - **`render()`**
  - **`componentDidMount()`**
- Updating:

  - `getDerivedStateFromProps()`
  - `shouldComponentUpdate()`
  - `render()`
  - `getSnapshotBeforeUpdate()`
  - `componentDidUpdate()`
- Unmounting:
  - `componentWillUnmount()`

Here is an example of a React component with lifecycle methods:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0,
    };
  }

  componentDidMount() {
    // This method is called after the component is inserted into the
DOM.
    console.log('componentDidMount');
  }

  componentDidUpdate(prevProps, prevState) {
    // This method is called after the component is updated.
    console.log('componentDidUpdate');
  }

  componentWillUnmount() {
    // This method is called before the component is removed from the
DOM.
    console.log('componentWillUnmount');
  }

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={() => this.setState({ count: this.state.count
+ 1 })}>
          Increment
        </button>
      </div>
    );
  }
}
```
Use code with caution.

Learn more

In this example, the **`componentDidMount()`** method is called after the component is inserted into the DOM. The **`componentDidUpdate()`** method is called after the component is updated. And the **`componentWillUnmount()`** method is called before the component is removed from the DOM.

You can use lifecycle methods to perform tasks such as:

- Setting up event listeners
- Fetching data
- Cleaning up resources

Lifecycle methods are a powerful tool that can help you write more efficient React components.

Here is an example of a React Hook:

```
function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
```
Use code with caution.

Learn more

This code creates a Counter component that uses the useState hook to manage its state. The useState hook returns an array of two values: the current state value and a function to update the state value. In this case, the state value is the number of times the button has been clicked. The update function is used to increment the count when the button is clicked.

The Counter component renders the current count and a button. When the button is clicked, the onClick event handler calls the update function to increment the count. The component then re-renders with the updated count.

Here are some other React Hooks:

- useEffect: This hook is used to perform side effects, such as fetching data or updating the DOM.
- useCallback: This hook is used to memoize callback functions.
- useLayoutEffect: This hook is used to perform layout effects, such as setting the height of an element.
- useRef: This hook is used to create references to DOM elements.
- useContext: This hook is used to access context values.

React Hooks are a powerful way to manage state and side effects in React components. They make it easier to write reusable and maintainable code.