React Routing

React Router is a powerful library for managing navigation and routing in React applications. It enables the creation of Single Page Applications (SPAs), where navigation to different pages doesn't require a full page reload. Instead, the content is dynamically rendered based on the URL, providing a smoother and faster user experience.

Single Page Applications (SPAs)

Single Page Applications (SPAs): An SPA is a web application that loads a single HTML page and dynamically updates the content as the user interacts with the app. This is achieved without reloading the entire page, leading to a more fluid user experience. React Router plays a crucial role in managing the navigation and rendering of components in SPAs.

Setting Up React Router

To use React Router in your React project, you need to install the react-router-dom package.

```
npm install react-router-dom
```

react-router vs react-router-dom

react-router

Core Library: react-router is the core library that provides the basic routing components and functionality. It is platform-agnostic, meaning it can be used in different environments like web, native mobile, or even server-side rendering.

Components and Functions: The core package includes essential components and functions like Router, Route, Switch, Redirect, useHistory, useLocation, useParams, etc.

Installation: Usually, you don't need to install react-router separately as it is a dependency of other platform-specific packages like react-router-dom.

react-router-dom

Web-Specific: react-router-dom is a package built on top of react-router specifically for web applications. It includes all the core functionality provided by react-router along with additional components and utilities designed for web browsers.

Additional Components: react-router-dom includes components like BrowserRouter, HashRouter, Link, NavLink, and more, which are tailored for web environments.

Installation: When setting up routing in a React web application, you typically install react-router-dom, which automatically includes react-router as a dependency.

Basic Setup

Import Required Components: Import the necessary components from react-router-dom such as BrowserRouter, Routes, Route and Link.

Define Routes: Use the Route component to define paths and associate them with components.

Use BrowserRouter: Wrap your application with the BrowserRouter component to enable routing.

Navigation Links: Use the Link component for navigation between different routes without refreshing the page.

Example

Let's create a basic example to demonstrate React Router setup and usage.

Step 1: Create React Components

Let's create a basic example to demonstrate React Router setup and usage.

src/Home.js

```
import React from 'react';

function Home() {
  return (
    <div>
      <h2>Home</h2>
      <p>Welcome to the Home Page!</p>
    </div>
  );
}

export default Home;
```

src/Aboutus.js

```
import React from 'react';

function Aboutus() {
  return (
    <div>
      <h2>About</h2>
      <p>Welcome to the About Page!</p>
    </div>
  );
}

export default Aboutus;
```

src/Contactus.js

```
import React from 'react';

function Contactus() {
  return (
    <div>
      <h2>Contact</h2>
      <p>Welcome to the Contact Page!</p>
    </div>
  );
}

export default Contactus;
```

src/PageNotFound.js

```
import React from 'react';

function PageNotFound() {
```

```
  return (
    <div>
      <p>PageNotFound!</p>
    </div>
  );
}

export default PageNotFound;
```

src/AboutCSE.js

```
import React from 'react';

function AboutCSE() {
  return (
    <div>
      <h2>About CSE</h2>
      <p>Welcome to the CSE Page!</p>
    </div>
  );
}

export default AboutCSE;
```

src/VisionMission.js

```
import React from 'react';

function VisionMission() {
  return (
    <div>
      <h2>AVision and Mission</h2>
      <p>Welcome to the Vision and Mission Page!</p>
    </div>
  );
}

export default VisionMission;
```

src/Elite.js

```
import React from 'react';

function VisionMission() {
  return (
    <div>
      <h2>Elite</h2>
      <p>Welcome to the Elite Page!</p>
    </div>
  );
}

export default Elite;
```

src/Navbar.js

```
import React from 'react';

function Navbar(props) {
    return (
        <div>
            <ul class="nav justify-content-center  ">
                <li class="nav-item">
                    <a   class="nav-link   active"   href="/"   aria-
current="page">Home</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="/aboutus">Aboutus</a>
                </li>
                <li class="nav-item">
                    <a                              class="nav-link"
href="/contactus">Contactus</a>
                </li>
                <li class="nav-item">
                    <a    class="nav-link"    href="/aboutcse">About
CSE</a>
                </li>
            </ul>
        </div>
    );
}

export default Navbar;
```

Step 2: Set Up Routing in App Component

Update the App.js file to set up routing.

src/App.js

```
import './App.css';
import AboutCSE from './components/AboutCSE';
import Aboutus from './components/Aboutus';
import Contactus from './components/Contactus';
import Home from './components/Home';
import {Routes, Route, Link} from 'react-router-dom';
import PageNotFound from './components/PageNotFound';
import Navbar from './components/Navbar';
import VisionMission from './components/VisionMission';
import Elite from './components/Elite';

function App() {
  return (
    <div className="App">
      <h1> Learning React Routers</h1>
      <Navbar/>
      <Routes>
        <Route path="" element={<Home/>}/>
        <Route path="/aboutus" element={<Aboutus/>}/>
        <Route path="/contactus" element={<Contactus/>}/>
        <Route path="/aboutcse" element={<AboutCSE/>}>
          <Route path="/aboutcse/vs" element={<VisionMission/>}/>
          <Route path="/aboutcse/elite" element={<Elite/>}/>
        </Route>
```

```
        <Route path="*" element={<PageNotFound/>}/>
      </Routes>
    </div>
  );
}

export default App;
```

<Navbar />: A custom component to display the navigation bar.

<Routes>: The Routes component is a container for all the Route components and switches between them based on the current URL.

<Route path="" element={<Home />} />: The default route that renders the Home component when the path is empty (i.e., the root path).

<Route path="/aboutus" element={<Aboutus />} />: A route that renders the Aboutus component when the URL path is /aboutus.

<Route path="/contactus" element={<Contactus />} />: A route that renders the Contactus component when the URL path is /contactus.

<Route path="/aboutcse" element={<AboutCSE />}>: A nested route that renders the AboutCSE component and has its own nested routes.

Nested Routes:

<Route path="/aboutcse/vs" element={<VisionMission />} />: A nested route that renders the VisionMission component when the URL path is /aboutcse/vs.

<Route path="/aboutcse/elite" element={<Elite />} />: A nested route that renders the Elite component when the URL path is /aboutcse/elite.

<Route path="*" element={<PageNotFound />} />: A catch-all route that renders the PageNotFound component for any undefined routes (i.e., a 404 page).

Passing and Extracting Route/Query Parameters in React Router

In React Router, route parameters and query parameters can be passed and extracted to make your application more dynamic. Here's a detailed explanation with examples:

Passing Route Parameters

Route parameters are parts of the URL that can change and are often used to represent specific items in a list, such as a user ID or product ID.

Defining a Route with Parameters

To define a route with a parameter, use the colon (:) syntax in the path.

App.js

```
import React from 'react';
import { BrowserRouter as Router, Route, Routes, Link } from 'react-
router-dom';
import Home from './components/Home';
import User from './components/User';

function App() {
  return (
    <Router>
      <div>
        <nav>
          <Link to="/">Home</Link>
          <Link to="/user/1">User 1</Link>
          <Link to="/user/2">User 2</Link>
        </nav>
        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/user/:userId" element={<User />} />
        </Routes>
      </div>
    </Router>
  );
}

export default App;
```

Extracting Route Parameters

To extract the parameters in the component, use the useParams hook from react-router-dom.

User.js

```
import React from 'react';
import { useParams } from 'react-router-dom';

function User() {
  let { userId } = useParams();
  return (
    <div>
      <h2>User ID: {userId}</h2>
    </div>
  );
}

export default User;
```

Passing Query Parameters

Query parameters are part of the URL after the ? and can be accessed using the URLSearchParams API.

Adding Query Parameters to Links

You can add query parameters directly in the Link component or through navigation.

App.js

```
import React from 'react';
import { BrowserRouter as Router, Route, Routes, Link } from 'react-
router-dom';
import Home from './components/Home';
import Search from './components/Search';

function App() {
  return (
    <Router>
      <div>
        <nav>
          <Link to="/">Home</Link>
          <Link to="/search?query=react">Search React</Link>
          <Link to="/search?query=router">Search Router</Link>
        </nav>
        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/search" element={<Search />} />
        </Routes>
      </div>
    </Router>
  );
}

export default App;
```

Extracting Query Parameters

To extract query parameters in the component, use the useLocation hook from react-router-dom and the URLSearchParams API.

Search.js

```
import React from 'react';
import { useLocation } from 'react-router-dom';

function useQuery() {
  return new URLSearchParams(useLocation().search);
}

function Search() {
  let query = useQuery();
  let searchQuery = query.get('query');
  return (
    <div>
      <h2>Search Query: {searchQuery}</h2>
    </div>
  );
}
```

```
export default Search;
```

Full Example with Both Route and Query Parameters

Let's combine both route and query parameters in a single example.

App.js

```
import React from 'react';
import { BrowserRouter as Router, Route, Routes, Link } from 'react-
router-dom';
import Home from './components/Home';
import User from './components/User';
import Search from './components/Search';

function App() {
  return (
    <Router>
      <div>
        <nav>
          <Link to="/">Home</Link>
          <Link to="/user/1">User 1</Link>
          <Link to="/user/2">User 2</Link>
          <Link to="/search?query=react">Search React</Link>
          <Link to="/search?query=router">Search Router</Link>
        </nav>
        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/user/:userId" element={<User />} />
          <Route path="/search" element={<Search />} />
        </Routes>
      </div>
    </Router>
  );
}

export default App;
```

User.js

```
import React from 'react';
import { useParams } from 'react-router-dom';

function User() {
  let { userId } = useParams();
  return (
    <div>
      <h2>User ID: {userId}</h2>
    </div>
  );
}

export default User;
```

Search.js

```
import React from 'react';
import { useLocation } from 'react-router-dom';

function useQuery() {
  return new URLSearchParams(useLocation().search);
}
```

```
function Search() {
  let query = useQuery();
  let searchQuery = query.get('query');
  return (
    <div>
      <h2>Search Query: {searchQuery}</h2>
    </div>
  );
}

export default Search;
```

Conclusion:

Passing Route Parameters: Define routes with parameters using the colon (:) syntax.

Extracting Route Parameters: Use the useParams hook to extract route parameters in a component.

Passing Query Parameters: Add query parameters to URLs directly or through navigation.

Extracting Query Parameters: Use the useLocation hook and URLSearchParams API to extract query parameters in a component.

This approach allows you to create dynamic and flexible routes in your React application, enhancing the user experience by providing specific data based on the URL parameters.

HTTP requests: Axios package, fetching data

HTTP (Hypertext Transfer Protocol) requests, are the fundamental way in which data is exchanged between a client and a server on the web. They are used to request resources or send data to a web server. HTTP is the protocol that underlies all data communication on the World Wide Web, and HTTP calls are how clients (such as web browsers or applications) interact with servers.

Types of HTTP Methods

There are several HTTP methods, each serving a different purpose:

GET:

- Used to request data from a specified resource.
- Requests using GET should only retrieve data and have no other effect.
- Example: Fetching a webpage or a list of items from an API.

POST:

- Used to send data to a server to create/update a resource.
- The data sent to the server with POST is stored in the request body.
- Example: Submitting form data or uploading a file.

PUT:

- Used to send data to a server to create or update a resource.
- The difference from POST is that PUT requests are idempotent; if you make the same request multiple times, the result will be the same as making it once.
- Example: Updating a user's information.

DELETE:

- Used to delete a specified resource.
- Example: Removing a user from a database.

PATCH:

- Used to apply partial modifications to a resource.
- Example: Updating a specific field in a user's profile.

OPTIONS:

- Used to describe the communication options for the target resource.
- Example: Checking which HTTP methods are supported by a server.

HEAD:

- Similar to GET, but it retrieves only the headers and not the body of the response.
- Example: Checking if a resource exists by examining the headers.

Making HTTP Calls in JavaScript

In JavaScript, HTTP calls can be made using various methods, including the Fetch API and libraries like Axios.

Using Fetch API

Fetch is a built-in JavaScript API designed specifically for making network requests in web browsers, offering a native way to fetch resources from the network. Fetch supports various request and response types, including JSON, FormData, and Blob.

Using Axios

Axios is a promise-based third-party HTTP client library for the browser and Node.js. It provides an easy-to-use API and offers features such as making asynchronous requests, handling request and response interception, and automatic transformation of JSON data.

Both Axios and the Fetch API are used to make HTTP requests in JavaScript applications. However, they have some differences in terms of features, ease of use, and flexibility. Here's a detailed comparison.

Feature	Axios	Fetch
Syntax	Clear and concise, with a chaining API	Requires additional processing for JSON
Browser support	Supported in all browsers	Supported in modern browsers
Cancellation	Supports request cancellation	No build-in support for cancellation
Error Handling	Axios has built-in error handling that catches HTTP errors and other issues.	Fetch does not automatically reject HTTP errors; you need to check and handle them explicitly.
Data Conversion	Automatically transforms JSON data when sending or receiving requests.	Requires explicit conversion to JSON using response.json().
Dependency	Requires installing an additional library (axios).	Built into modern browsers, no additional library required.

Lets look into examples for each HTTP requests

HTTP GET method:

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

const HttpRequest = () => {
     let [users, setUsers] = useState([]);

    useEffect(() => {
        axios.get("https://reqres.in/api/users?page=2")
        .then(response => {
            let mydata = response.data.data;
            setUsers(mydata);
        })
        .catch(error => {
            console.error('There was an error making the request!',
error);
        });
    }, []); // Empty dependency array ensures this runs only once

    return (
        <div>
            <h1>HTTP requests from React using Axios</h1>
            <table>
            { users.map((obj) => (
```

```
              <tr key={obj.id}>
                <td>{obj.id}</td>
                <td>{obj.email}</td>
                <td><img src={obj.avatar} alt="Avatar"/></td>
              </tr>
            ))}
            </table>
        </div>
    );
};

export default HttpRequest;
```

Detailed Breakdown

1. State Initialization:

```
let [users, setUsers] = useState([]);
```

useState([]): Initializes a state variable users as an empty array and provides a function setUsers to update the state.

2. Fetching Data:

```
 useEffect(() => {
        axios.get("https://reqres.in/api/users?page=2")
        .then(response => {
            let mydata = response.data.data;
            setUsers(mydata);
        })
        .catch(error => {
            console.error('There was an error making the request!',
error);
        });
    }, []); // Empty dependency array ensures this runs only once
```

axios.get: Makes a GET request to the specified URL (https://reqres.in/api/users?page=2).

.then(response => { ... }): Handles the response from the server.

response.data.data: Accesses the user data from the response.

setUsers(mydata): Updates the users state with the fetched data.

3.Rendering the Component:

```
return (
    <div>
        <h1>HTTP requests from React using Axios</h1>
        <table>
        { users.map((obj) => (
          <tr key={obj.id}>
            <td>{obj.id}</td>
            <td>{obj.email}</td>
            <td><img src={obj.avatar} alt="Avatar"/></td>
          </tr>
        ))}
        </table>
    </div>
);
```

<h1>HTTP requests from React using Axios</h1>: Displays a heading.

<table>: Renders a table to display the user data.

users.map((obj) => (...)): Iterates over the users array and renders a row for each user.

key={obj.id}: Each child in a list should have a unique "key" prop to help React identify which items have changed.

<td>{obj.id}</td>: Displays the user's ID.

<td>{obj.email}</td>: Displays the user's email.

<td></td>: Displays the user's avatar image with an alt attribute for accessibility.

HTTP POST method:

HttpPost Component

The HttpPost component makes a POST request to the https://reqres.in/api/users endpoint and updates the component's state with the ID returned from the response.

```
import axios from 'axios';
import React, { useEffect, useState } from 'react';
import HttpRequest from './HttpRequest';

const HttpPost = () => {
    let [postID, setPostId] = useState(0);

    const user = {
        "name": "morpheus",
        "job": "leader"
    };

    useEffect(() => {
        axios.post('https://reqres.in/api/users', user)
            .then(response => {
                console.log(response.data); // Log the response data
                setPostId(response.data.id);
            })
            .catch(error => {
                console.error('There   was   an   error   making   the
request!', error);
            });
    }, []);

    return (
        <div>
            <h1> This is post Request</h1>
            <h4> {postID} </h4>
            {/* <HttpRequest/> */}
        </div>
    );
};

export default HttpPost;
```

Detailed Breakdown

1. State Initialization:

```
let [postID, setPostId] = useState(0);
```

useState(0): Initializes a state variable postID with the initial value 0 and provides a function setPostId to update the state.

2. User Object:

```
const user = {
    "name": "morpheus",
    "job": "leader"
};
```

Defines a user object that will be sent in the POST request body.

3. useEffect Hook:

```
useEffect(() => {
    axios.post('https://reqres.in/api/users', user)
        .then(response => {
            console.log(response.data); // Log the response data
            setPostId(response.data.id);
        })
        .catch(error => {
            console.error('There was an error making the request!',
error);
        });
}, []);
```

useEffect: A hook that runs the provided function after the component mounts.

axios.post: Makes a POST request to the specified URL (https://reqres.in/api/users) with the user object as the request body.

.then(response => { ... }): Handles the successful response from the server.

response.data: The data returned from the server.

setPostId(response.data.id): Updates the postID state with the ID returned in the response.

.catch(error => { ... }): Handles any errors that occur during the request.

console.error: Logs the error to the console.

The empty dependency array [] ensures that this effect runs only once, after the initial render.

4. Rendering the Component:

```
return (
    <div>
        <h1> This is post Request</h1>
        <h4> {postID} </h4>
    </div>
);
```

<h1> This is post Request</h1>: Displays a heading.

<h4> {postID} </h4>: Displays the postID state, which is updated with the ID returned from the POST request.

HTTP PUT method:

The HttpPut component makes a PUT request to the https://reqres.in/api/articles/1 endpoint and updates the component's state with the updatedAt field from the response data.

The HttpPut component demonstrates how to make an HTTP PUT request using the Axios library in a React functional component. Here's a detailed explanation of the code:

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

const HttpPut = () => {
    let [updatedAt, setUpdatedAt] = useState(0);

    const user = {
        "name": "morpheus",
        "job": "leader"
    };

    useEffect(() => {
        axios.put('https://reqres.in/api/articles/1', user)
            .then(response => {
                console.log(response.data); // Log the response data
                setUpdatedAt(response.data.updatedAt);
            })
            .catch(error => {
                console.error('There was an error making the PUT!',
error);
            });
    }, []);

    return (
        <div>
            <h1>HTTP PUT Example</h1>
            {updatedAt}
        </div>
    );
};

export default HttpPut;
```

Detailed Breakdown

1. State Initialization:

```
let [updatedAt, setUpdatedAt] = useState(0);
```

useState(0): Initializes a state variable updatedAt with the initial value 0 and provides a function setUpdatedAt to update the state.

2. User Object:

```
const user = {
    "name": "morpheus",
    "job": "leader"
};
```

Defines a user object that will be sent in the PUT request body.

3. useEffect Hook:

```
useEffect(() => {
    axios.put('https://reqres.in/api/articles/1', user)
        .then(response => {
            console.log(response.data); // Log the response data
            setUpdatedAt(response.data.updatedAt);
        })
        .catch(error => {
            console.error('There was an error making the PUT!',
error);
        });
}, []);
```

useEffect: A hook that runs the provided function after the component mounts.

axios.put: Makes a PUT request to the specified URL (https://reqres.in/api/articles/1) with the user object as the request body.

.then(response => { ... }): Handles the successful response from the server.

response.data: The data returned from the server.

setUpdatedAt(response.data.updatedAt): Updates the updatedAt state with the updatedAt field from the response.

.catch(error => { ... }): Handles any errors that occur during the request.

console.error: Logs the error to the console.

The empty dependency array [] ensures that this effect runs only once, after the initial render.

4. Rendering the Component:

```
return (
    <div>
        <h1>HTTP PUT Example</h1>
        {updatedAt}
    </div>
);
```

<h1>HTTP PUT Example</h1>: Displays a heading.

{updatedAt}: Displays the updatedAt state, which is updated with the updatedAt field from the PUT request response.

HTTP DELETE method:

The HttpDelete component makes a DELETE request to the https://reqres.in/api/posts/1 endpoint and updates the component's state with a status message indicating whether the delete operation was successful.

The HttpDelete component demonstrates how to make an HTTP DELETE request using the Axios library in a React functional component. Here's a detailed explanation of the code:

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

const HttpDelete = () => {
    let [deleteStatus, setDeleteStatus] = useState();

    useEffect(() => {
        axios.delete('https://reqres.in/api/posts/1')
```

```
            .then(response => {
                setDeleteStatus('Delete successful');
                console.log(response); // Log the response data
            })
            .catch(error => {
                console.error('There was an error making the DELETE
request!', error);
            });
    }, []);

    return (
        <div>
            <h1>HTTP DELETE Example</h1>
            <h3>{deleteStatus}</h3>
        </div>
    );
};

export default HttpDelete;
```

Detailed Breakdown

1. State Initialization:

```
let [deleteStatus, setDeleteStatus] = useState();
```

useState(): Initializes a state variable deleteStatus and provides a function setDeleteStatus to update the state. Initially, deleteStatus is undefined.

2. useEffect Hook:

```
useEffect(() => {
    axios.delete('https://reqres.in/api/posts/1')
        .then(response => {
            setDeleteStatus('Delete successful');
            console.log(response); // Log the response data
        })
        .catch(error => {
            console.error('There was an error making the DELETE
request!', error);
        });
}, []);
```

useEffect: A hook that runs the provided function after the component mounts.

axios.delete: Makes a DELETE request to the specified URL (https://reqres.in/api/posts/1).

.then(response => { ... }): Handles the successful response from the server.

setDeleteStatus('Delete successful'): Updates the deleteStatus state with the message 'Delete successful'.

console.log(response): Logs the response data to the console.

.catch(error => { ... }): Handles any errors that occur during the request.

console.error('There was an error making the DELETE request!', error): Logs the error to the console.

The empty dependency array [] ensures that this effect runs only once, after the initial render.

3. Rendering the Component:

```
return (
    <div>
        <h1>HTTP DELETE Example</h1>
        <h3>{deleteStatus}</h3>
    </div>
);
```

<h1>HTTP DELETE Example</h1>: Displays a heading.

<h3>{deleteStatus}</h3>: Displays the deleteStatus state, which is updated with the message 'Delete successful' if the DELETE request is successful.