**React JS**

React is a framework that employs Webpack to automatically compile React, JSX, and ES6 code while handling CSS file prefixes. React is a [JavaScript](https://www.simplilearn.com/tutorials/javascript-tutorial/introduction-to-javascript)-based UI development library. Although React is a library rather than a language, it is widely used in web development. The library first appeared in May 2013 and is now one of the most commonly used frontend libraries for web development.

**Section 21: SPA with React Router**

**Q. What is Routing?**

In a single page application routing refers to the process of navigating between different pages without triggering a full page reload. The application initially loads a single page HTML.

**Q. What is Single Page Application?**

A Single Page Application (SPA) is a web application that loads a single HTML page and dynamically updates the content as the user interacts with the application.

For use the routing we have to install a package

npm install react-router-dom

This package allows us listen to URL changes and then load different content

**Adding Routing in the application**

* The first step is that we must define the routes we want to support, so we must define which URLs paths we want to support and which components should be loaded for different paths.
* The second step is to activate our router and load the route definitions that we defined in the first step.
* the third step is to make sure that we have all these components that we want to load and that maybe also provide some means of navigating between those pages so that our users can move smoothly

**Implementation**

* Import **createBrowserRouter** from **react-router-dom** which allows us to define our routes.
* Now call this function and pass an array of route definition objects. This object will contain the **path** and the **element** which render when we enter the path in browser.
* We have to store the return value of **createBrowserRouter** functionina constant to tell React that’s this router should be render to the screen

*const* router = createBrowserRouter([

  {path:'/', element:<*HomePage*/>},{path:'/about', element:<*AboutPage*/>}

])

* Tell React that this router should be use. We need to import **RouterProvider** component from **react-router-dom**.
* This **RouterProvider** will take props with the name of **router** which we must set.

*function* App() {

  return <*RouterProvider* router={router}/>

}

* Now if we run the application we can access this path.

We can also use the **createRoutesFromElements** function from react-router-dom for doing the same task.

*const* routerDefinition = createRoutesFromElements(

  <*Route*>

    <*Route* path='/' element={<*HomePage*/>}/>

    <*Route* path='/' element={<*AboutPage*/>}/>

  </*Route*>

)

*const* router = createBrowserRouter(routerDefinition);

*function* App() {

return <*RouterProvider* router={router}/>

}

At the moment, we always have to manually edit the URL if we want to navigate from the starting page to the products page and vice versa. And it's, of course, not very realistic.

We resolve this problem by providing the <a/> tag but in anchor tag there is one problem, when we click on a link it will refresh the page and reload the whole application again.

*function* HomePage() {

  return <>

    <div>This is a HomePage</div>

    <p>This is a paragraph <a href='/about'>Go to the about page</a></p>

    </>

}

This thing is happening because we are sending a new request to the server which serving this website. And because of this the whole application will load again and it causes a big problem, it will slow down the application.

For resolving this problem we will use another component call **Link** which provided by **react-router-dom**. It stops the reloading of the application and page won’t get the reload, so it will increase the performance of the application.

*function* HomePage() {

    console.log('home page render');

  return <>

    <div>This is a HomePage</div>

    <p>This is a paragraph

        <*Link* to='/about'>

            Go to the about page

        </*Link*>

    </p>

    </>

}

Now we might want to add a navigation bar at the top, which actually lets us navigate between the homepage and the about page.

*function* MainNavigation() {

  return (

    <header>

        <nav>

            <ul>

                <li>

                    <*Link* to='/'>Home</*Link*>

                </li>

                <li>

                    <*Link* to='/'>About</*Link*>

                </li>

            </ul>

        </nav>

    </header>

  )

}

These links only work if they rendered, so have to put these links in the **RouterProvider** function. We can’t add this **MainNavigation** next to RouterProvider.

With **react-router-dom**, you add an extra route to your route definitions and you use a path of slash nothing. And then here, you add an element that actually loads the layout wrapper

That should be wrapped around the other routes.

* Create a **RootLayout** component.

*function* RootLayout() {

  return <*MainNavigation* />

}

* And now edit the content of createBrowserRouter() function in App.js

*const* router = createBrowserRouter([

  {

    path: "/",

    element: <*RootLayout* />,

    children: [

      { path: "/", element: <*HomePage* /> },

      { path: "/about", element: <*AboutPage* /> },

    ],

  },

]);

RootLayout component will work here as a wrapper to these routes.

Here RootLayout is a parent routes of ‘/’ and ‘/about’. Now we run application so RootLayout component will be render and then

*function* RootLayout() {

  return (

    <>

      <*MainNavigation* />

<Outlet/>

    </>

  );

}

MainNavigation component will render which show the links in browser.

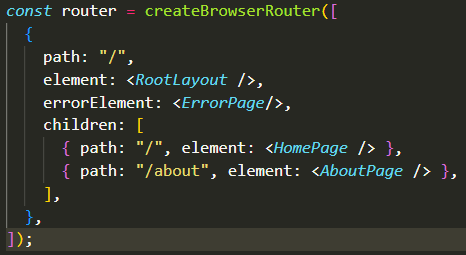
**Outlet**: This component marks the place where the child route element should render.

**Error Page**

If the user pass wrong url which is not exist in react-routing on that case react-router-dom will show the default error page. We can create our own customize error page.

**Steps:**

* Create a ErrorPage component and add the error code in this component
* Then add this error page in createBrowserRouter() method



**Working with Navigation Links**

We can use NavLink in place of Link for designing purpose. NavLink is also provided by **react-router-dom** package.

<ul>

<li>

            <*NavLink* to="/"

              className={({ *isActive* }) *=> // this function is provided by NavLink*

*isActive* ? classes.active : undefined

              }

            >

// Note: We can also use this function in **style property**

              Home

            </*NavLink*>

          </li>

          <li>

          <*NavLink*

              to="/about"

              className={({ *isActive* }) *=>*

*isActive* ? classes.active : undefined

              }

            >

              About

            </*NavLink*>

          </li>

</ul>

We can apply styling on <a> tag because by default react-router-dom is uses <a> tag for <Link> and <NavLink>

.list a:hover,

  .list a.active {

*color*: var(--color-primary-800);

*text-decoration*: underline;

  }

**Imperative Routing**

Right now, we allow users to navigate between our pages by providing links which can be clicked, this is the default way of providing navigation to users, but it's not the only way. In some situations, for example, maybe because of some forum was submitted, or because some timer expired, you might want to trigger a navigation action from inside Code, and you can do this by using another special feature provided by react-router-dom.

**useNavigate hook**

With the help of useNavigate hook we can programmatically implement the routing

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

import React from 'react'

*function* HomePage() {

*const* navigate = useNavigate();

*function* navigateHandler(){

    navigate('/about');

  }

  return <>

    <div>This is a HomePage</div>

    <p>

      <button onClick={navigateHandler}>Navigate to About Page</button>

    </p>

    </>

}

**Dynamic path segment or Path parameter**

In React Router, both "Dynamic path segments" and "Path parameters" refer to the concept of defining routes with dynamic paths, which allow your application to match and handle URLs dynamically.

Dynamic path segments in React Router refer to defining routes with parts of the URL path that can change dynamically. For example, consider a route like **/users/:userId**, where **:userId** is a dynamic segment that can represent any user ID. This allows you to match URLs like **/users/123**, **/users/456**, etc. The value of **userId** can then be accessed within your components using the **useParams** hook (for functional components) or **this.props.match.params** (for class components) provided by React Router.

*const* router = createBrowserRouter([

  {

    path: "/",

    element: <*RootLayout* />,

    errorElement: <*ErrorPage*/>,

    children: [

      { path: "/", element: <*HomePage* /> },

      { path: "/about", element: <*AboutPage* /> },

      { path: "/product", element: <*ProductPage* /> },

      { path: "/product/:productId", element: <*ProductDetails*/>}

    ],

  },

]);

*const* PRODUCTS = [

  { id: "p1", title: "Product-1" },

  { id: "p2", title: "Product-2" },

  { id: "p3", title: "Product-3" },

];

*function* ProductPage() {

  return (

    <>

      <div>This is a ProductPage</div>

      <ul>

        {PRODUCTS.map((*product*) *=>* (

          <li key={*product*.id}>

            <*Link* to={"/product/" + *product*.id} style={{color: "white"}}>{*product*.title}</*Link*>

          </li>

        ))}

      </ul>

    </>

  );

}

*function* ProductDetails() {

*const* param = useParams();

  return <>

    <div>ProductDetails</div>

    <h1>{param.productId}</h1>

  </>

}

**Absolute and Relative path**

In React Router, both absolute and relative paths are used to define the routing behavior of your application, determining where a particular route should load based on its path definition.

* **Absolute Paths:** Use absolute paths when you want to define route that are independent of the current location and should always match a specific URL.

<*Route* path="/users" component={Users} />

* **Relative Paths:** Use relative paths when you want to define routes that depend on the current location, especially in nested routing scenarios where you want to append paths to the current URL dynamically.

<*Route* path={`${match.url}/profile`} component={Profile} />

Note: if the path is starting with / it means it a absolute path

**Back Button**

Using **<Link to='..'>Back </Link>** in React Router DOM is a common approach to create a "Back" button or link that navigates to the previous page in the browser's history. When you specify **to='..'**, it means to navigate up one level in the URL hierarchy.

*function* ProductDetails() {

*const* param = useParams();

  return <>

    <div>ProductDetails</div>

    <h1>{param.productId}</h1>

    <p>

       <*Link* to='..' relative='path' style={{color:'white'}}>Back</*Link*>

    </p>

  </>

}

**Index: true**

*const* router = createBrowserRouter([

  {

    path: "/",

    element: <*RootLayout* />,

    errorElement: <*ErrorPage*/>,

    children: [

      { index: true,  element: <*HomePage* /> },

      { path: "/about", element: <*AboutPage* /> },

      { path: "/product", element: <*ProductPage* /> },

      { path: "/product/:productId", element: <*ProductDetails*/>}

    ],

  },

]);

**New Project: Advance React-router-dom**

Run backend: npm start

**Data Fetching and Submission**

Most important feature provided by React router is related to data fetching and submission.

A loader in react-router is a function that is used to fetch data for a route before it is rendered. When you click on a link which directs you to a route, the loader function runs and gets the data ready for the route before it renders.

path: "events",

        element: <*EventRoot* />,

        children: [

          {

            index: true,

            element: <*EventPage* />,

            loader: async () *=>* {

*const* response = await fetch("http://localhost:8080/events");

              if (!response.ok) {

                // error handling code

              } else {

*const* resData = await response.json();

                return resData.events;

              }

            },

          },

          { path: ":id", element: <*EventDetailPage* /> },

          { path: "new", element: <*NewEventPage* /> },

Now the fetched data will be available for the **EventPage** component.

With the help of **useLoaderData** hooks we can fetch the data in **EventPage. useLoaderData** hooks provided by react-router-dom.

This hook will access the **closest loader data**.

import EventsList from '../components/EventsList';

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

*function* EventsPage() {

*const* events = useLoaderData();

  return <*EventsList* events={events} />

}

export default EventsPage;

We can access the data in child level components of **EventPage**

*function* EventsList() {

*const* events = useLoaderData();

Here we are accessing the data in **EventsList** component because it’s a child component of EventPage.

We have to put this data fetching code in EventPage and access it in App.js. It’s a good practice.

**EventPage.js**

export async *function* eventLoader(){

*const* response = await fetch("http://localhost:8080/events");

    if (!response.ok) {

      // error handling code

    } else {

*const* resData = await response.json();

      return resData.events;

    }

}

**App.js**

  {

        path: "events",

        element: <*EventRoot* />,

        children: [

          {

            index: true,

            element: <*EventPage* />,

            loader: eventLoader

          },

          { path: ":id", element: <*EventDetailPage* /> },

          { path: "new", element: <*NewEventPage* /> },

          { path: ":id/edit", element: <*EditEventPage* /> },

        ],

      },

**useNavigation**

useNavigation is a hook which gives access to navigation object. It's useful when you cannot pass the navigation prop into the component directly, or don't want to pass it in case of a deeply nested child.

useNavigation() returns the navigation prop of the screen it's inside. This hook is available in react-router-dom.

*function* RootLayout() {

*const* navigation = useNavigation();

  return <>

    <*MainNavigation*/>

    <main>

      {navigation.state === 'loading' && <p>data is loading...</p>}

      <*Outlet*/>

    </main>

  </>

}

Note: The code written inside the loader function is executed by the browser not by the server. We can add only those features inside this loader function which will be provided by the browser. Like **localStorage** etc. but we can’t use the react related features inside this loader like **useState** etc.

**Error handling in loader function**

**1st way**

export default *function* EventsPage() {

*const* events = useLoaderData();

  if(events.isError){

    return <p>{events.message}</p>

  }

  return <*EventsList* events={events} />

}

export async *function* loader(){

*const* response = await fetch("http://localhost:8080/events");

    if (!response.ok) {

      return {isError: true, message: 'Could not fetch the data'}

    } else {

*const* resData = await response.json();

      return resData.events;

    }

}

**2nd Way**

export async *function* loader(){

*const* response = await fetch("http://localhost:8080/eventssss");

    if (!response.ok) {

      throw {message:'Could not fetch the data'};

    } else {

*const* resData = await response.json();

      return resData.events;

    }

}

now create an **ErrorPage** and pass it inapp.js like

{

    path: "/",

    element: <*RootLayout* />,

    errorElement: <*ErrorPage*/>,

    children: [

      { index: true, element: <*HomePage* /> },

When the error occurred in loader function this ErrorPage will be render.

**3rd Way using useRouteError Hook**

useRouteError() Hook. The useRouteError() hook allows us to handle route errors in a functional component. It helps us handle and display errors that are thrown during an action() , loader() or render of a component. We'll go through a few examples to better explain how this works. This hook is provided by react-router-dom.

**Event.js**

export async *function* loader() {

*const* response = await fetch(“http://localhost:8080/events”);

  if (!response.ok) {

    throw new *Response*(

      JSON.stringify({ message: “Could not fetch the data” }),

      { status: 500 }

    );

  } else {

*const* resData = await response.json();

    return resData.events;

  }

}

Good to throw an Response() object as an error so we can customize error page.

ErrorPage.js

export default *function* ErrorPage() {

*const* error =  useRouteError();

*let* title = 'An error occurred';

*let* content = 'Something went wrong';

  if(error.status === 500){

    title = 'Event data not found';

    content = error.data.message;

  }

return <*PageContent* title={title}>

    <p>{content}</p>

  </*PageContent*>

}

**How to access id of an item in loader function**

By default when we use the loader function in route react-router-dom will pass two arguments to the loader function **request** and **params.**

With the help of params we can get the id of the url.

export async *function*  eventLoaderById({*request* ,*params*}){

*const* id = *params*.id;

*const* response = await fetch('http://localhost:8080/events/'+id);

    if(!response.ok){

      throw json({message:'Could not fetch the details for the selected event'}, {status:500})

// json is provided by react-router-dom

    }

    else{

      return response;

    }

}

**useRouteLoadData() hook**

This hook makes the data at any currently rendered route available anywhere in the tree. This is useful for components deep in the tree needing data from routes much farther up, as well as parent routes needing the data of child routes deeper in the tree.

export default *function* EventDetailPage() {

*const* data = useRouteLoaderData('event-data');

  return (

    <>

      <*EventItem* event = {data.event}/>

    </>

  );

}

**App.js**

{

            path: ":id",

            id: "event-data",

// mandatory, with the help of this id we can access the data in pages

            loader: eventLoaderById,

            children: [

              { index: true, element: <*EventDetailPage* /> },

              { path: "edit", element: <*EditEventPage* /> },

            ],

          },

          { path: "new", element: <*NewEventPage* /> },

In the above code EventDetailsPage and EditEventPage able to access the loaded data with the help if id, in the above example id is **event-data** so will pass this id in **useRouteLoaderData(event-data)** hook so it will return the data which associated with the passed it.

Note: loader and action function is executed by react-router-dom.

Learn about Form tag provided by react-router-dom

**action property**

action is also same as loader action also take the function. Loader is used for loading the data and action is used for adding the data.

Action code also execute in the browser so we can’t use any react code inside this function we only use the browser related code in it like localStorage.

**NewEventPage.js**

export async *function* action({*request*, *params*}){

*const* data = await *request*.formData();  // it will return the form data

*const* eventData = {

      title: data.get('title'), //get() method is use for getting the data from the form

      image: data.get('image'),

      date: data.get('date'),

      description: data.get('description')

    }

    console.log(eventData);

*const* response = await fetch('http://localhost:8080/events',{

      method:'POST',

      headers: {'Content-Type':'application/json'},

      body: JSON.stringify(eventData)

    })

    if(!response.ok){

      throw json({message: 'Could not save event'}, {status:500})

      // json is also provided by react-router-dom package

    }

    return redirect('/events')

}

App.js

{ path: "new", element: <*NewEventPage* />, action:action},

**redirect** is provided by react-router-dom package it’s useful when we post the request and we after request post we want to redirect to a another page on that case redirect is very useful function.

**useSubmit() hook**

The imperative version of <Form> that lets you, the programmer, submits a form instead of the user.

**Delete the Data programmatically**

EventDetailsPage.js

export async *function*  eventDeleteAction({*request*, *params*}){

*const* id = *params*.id;

*const* response = await fetch('http://localhost:8080/events/'+id,{

    method: *request*.method,

  });

  if(!response.ok){

    throw json({message:'Could not delete the event for the selected id'}, {status:500})

  }

  return redirect('/events');

}

EventItem.js

*const* submit = useSubmit();

*function* startDeleteHandler() {

*const* proceed = window.confirm('Are you sure!');

    if(proceed){

      submit(null,{method:'delete'})

      // first argument the data we want to submit

    }

  }

App.js

{ index: true, element: <*EventDetailPage* />, action : eventDeleteAction },