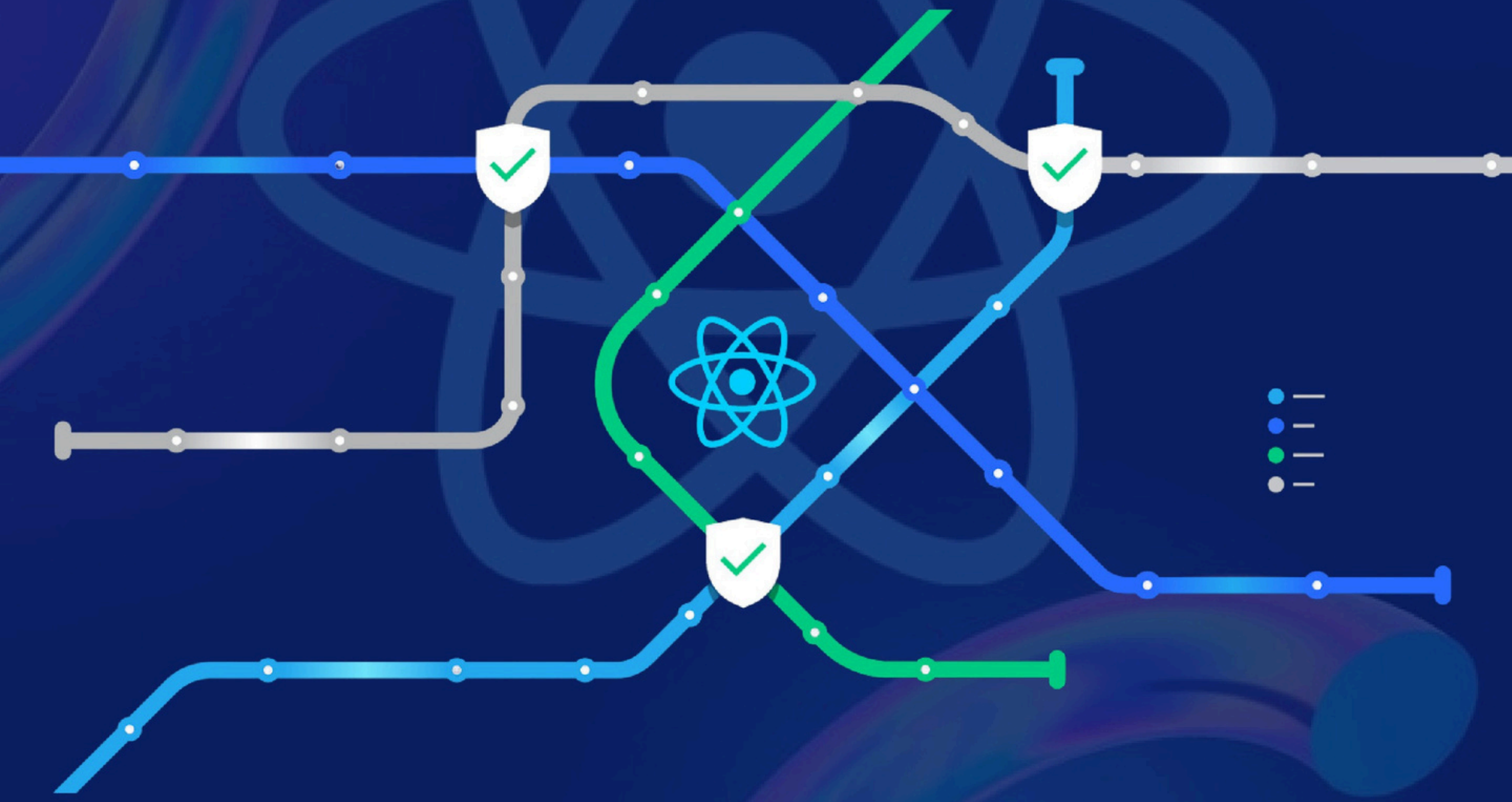
Routing In React



What is React Routing?

Routing in React is a way to navigate between different pages or views in a React application.

It allows you to build a **single-page application (SPA) with multiple sections or pages**, such as a homepage, about page, contact page, etc., without having to reload the whole page every time the user clicks a link. In traditional multi-page web applications,
navigation involved loading a new HTML page from the server. In contrast, React.js uses **client-side routing** to
update the content of the page dynamically without reloading the entire page.

Why Use Client-Side Routing?

- **Faster Navigation:** Only the content that changes is updated, leading to faster page transitions.

- **Smooth User Experience:** Client-side routing enables seamless navigation without full-page reloads, enhancing the user experience.

- **State Management:** You can maintain the state of your application during navigation without losing the current context.

React Router Overview

React Router is the most popular library for routing in React applications. It provides a simple API to define routes and map them to React components.

Key Concepts Of React router

- **Routes:** Define the mapping between a URL path and a React component.
- **Router:** The context provider that makes routing work in your application.
- **Link:** A component to navigate between routes.
- **Switch:** Renders the first route that matches the current URL.
- **Nested Routes:** Routes within routes to handle more complex applications.
-

How Does Routing Work in React?

1. React Router: To implement routing in React, we use a popular library called React Router. This library helps manage navigation and rendering of components based on the URL.

2. Routes: Routes define the different paths (URLs) of your application and the components that should be displayed for each path. For example:

- **“/”** might display the **Home** component.
- **“/about”** might display the About component.
- **“/contact”** might display the Contact component.

3. Links: React Router provides <Link> components that allow you to navigate between different routes without reloading the page. They work like traditional HTML links but keep the app fast.

Setting Up React Router

To use React Router, you need to install the library in your React project using the npm or yarn add commands. In my case I will be using npm:

```
kemil@DESKTOP-QKVDT62 MINGW64 /c/Desktop/My-Projects/my-vite-app
$ npm install react-router-dom
```

Once installed, you can import the necessary components from **react-router-dom** and set up routing in your app.

You typically define your routes in a **main file**, such as **App.jsx**.

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

const Home = () => <h2>Home Page</h2>;
const About = () => <h2>About Page</h2>;
const Contact = () => <h2>Contact Page</h2>;

const App = () => {
  return (
    <Router>
      <nav>
        <Link to="/">Home</Link>
        <Link to="/about">About</Link>
        <Link to="/contact">Contact</Link>
      </nav>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="/contact" element={<Contact />} />
      </Routes>
    </Router>
  );
};

export default App;

```

- **BrowserRouter (Router):** Wraps the application and provides routing capabilities.
- **<Link>:** Creates navigation links to different routes (Home, About, and Contact).
- **<Routes> and <Route>:** Defines the different paths (/, /about, /contact) and the components (**Home**, **About**, **Contact**) to be rendered when those paths are accessed.

Dynamic Routing with URL Parameters

React Router allows you to create **dynamic routes** with parameters that change based on the URL. This is useful for pages like user profiles, product details, etc.

Let's add a dynamic route to display a user profile based on the user ID.
Example of dynamic routing step-by-step:

Step 1: Add **Routes** with Dynamic Parameter

```jsx
// src/App.jsx

import UserProfile from './components/UserProfile';

// Add new route for user profile
<Route path="/user/:userId" element={<UserProfile />} />
```

Step 2: Create the UserProfile Component and add these codes:

```jsx
// src/components/UserProfile.jsx
import { useParams } from 'react-router-dom';

function UserProfile() {
  // Retrieve the dynamic parameter from the URL
  const { userId } = useParams();

  return <h2>User Profile: {userId}</h2>;
}

export default UserProfile;
```

- **useParams** is a hook provided by React Router to access the parameters in the URL.

Now, use the **Link** component to navigate to a dynamic route.

```
1 <Link to="/user/123">View User 123</Link>
```

When the link is clicked, it navigates to **/user/123** and renders the **UserProfile** component with userId set to 123 like this: **localhost:5173/user/123**

Nested Routes

Nested Routing in React is a technique that allows you to **define routes within other routes**, creating a hierarchy or "nesting" of routes. This is useful when you have a component that has its own set of sub-routes or pages.

Suppose you have a blog with a **main page** and subpages for different articles:

```jsx
// src/App.jsx

import Blog from './components/Blog';
import Article from './components/Article';

// Define nested routes
<Route path="/blog/*" element={<Blog />}>
  <Route path=":articleId" element={<Article />} />
</Route>
```

Create the Parent Component with **Outlet.**

Use the **Outlet** component from React Router to render the child components.

```jsx
// src/components/Blog.js
import { Outlet, Link } from 'react-router-dom';

function Blog() {
  return (
    <div>
      <h2>Blog</h2>
      <ul>
        <li><Link to="article1">Article 1</Link></li>
        <li><Link to="article2">Article 2</Link></li>
      </ul>
      <Outlet /> // the nested route will be displayed here
    </div>
  );
}

export default Blog;
```

- **Outlet:** A placeholder for nested routes to be rendered.
- **Link:** Used to navigate between the articles within the blog.

Programmatic Navigation

Sometimes, you need to navigate to a different route programmatically, such as after a form submission or a user action.

React Router provides the **useNavigate** hook to perform navigation.
Instead of relying on traditional **<Link>** components or **anchor tags** to navigate between pages, **useNavigate** provides a way to perform navigation through JavaScript code.
This is particularly useful in situations where you need to change the route based on certain conditions, such as after form submission, button clicks, or user actions.

Here’s a simple example to demonstrate how to use **useNavigate** in a React component:

```jsx
// src/components/Login.js
import { useNavigate } from 'react-router-dom';

function Login() {
  const navigate = useNavigate();

  const handleLogin = () => {
    // Perform login logic...
    navigate('/dashboard'); // Navigate to dashboard after login
  };

  return <button onClick={handleLogin}>Login</button>;
}

export default Login;
```

- The **useNavigate** hook is imported from the react-router-dom library.
- **const navigate = useNavigate();** assigns the **navigate** function to a variable. This function can now be used to perform navigation actions.
- Inside the **handleLogin** function, **navigate('/dashboard')** is called to programmatically navigate to the /dashboard page. This happens when the login logic is successful.

Protected Routes (Authentication)

Protected Routes in React are routes that require a user to be **authenticated (logged in)** before they can access certain parts of your application.

This is a common scenario in applications where some content or features should only be available to authenticated users, such as a user's profile, a dashboard, or any other secure content.

How to Implement Protected Routes in React

To implement protected routes in React, you typically use a combination of **React Router** and a **state management** solution (like React Context or Redux) to keep track of the user's authentication status.

Let's create an example of a simple React app with protected routes:

1. Define a component to handle protected routes:

```jsx
// src/ProtectedRoute.jsx
import { Navigate } from 'react-router-dom';

function ProtectedRoute({ isAuthenticated, children }) {
  if (!isAuthenticated) {
    return <Navigate to="/login" replace />;
  }
  return children;
}

export default ProtectedRoute;
```

- **Navigate:** This is a component from the react-router-dom library used to programmatically redirect users to a different route.
- **isAuthenticated:** A boolean value indicating whether the user is logged in (authenticated).
- **children:** Represents the child components that should be rendered if the user is authenticated. These children could be any component that requires protection, such as a dashboard, profile page, or other private routes.

- **!isAuthenticated** checks if the user is not authenticated .
- If the user is not authenticated, it renders the **Navigate component** from react-router-dom to redirect the user to the **/login** page.
- The **replace** attribute ensures that the login route will replace the current entry in the browser's history stack, preventing the user from navigating back to the protected route after logging in.

2. Use the ProtectedRoute component to wrap any route that needs protection:

```
// src/App.js

<Route
  path="/dashboard"
  element={
    <ProtectedRoute isAuthenticated={isLoggedIn}>
      <Dashboard />
    </ProtectedRoute>
  }
/>
```

When a user tries to access the /dashboard route:

- The **ProtectedRoute** component will check the isAuthenticated prop.
- If **isAuthenticated** is true (**isLoggedIn** is true), the **Dashboard** component will be rendered.
- If **isAuthenticated** is false (**isLoggedIn** is false), the user will be redirected to the **/login** page (or another specified route).

Handling 404 Pages (Not Found)

This is a good practice of managing situations when a user tries to access a route that doesn't exist in your application. This is commonly referred to as a "404 Page" or "Not Found" page.

Implementing a 404 page ensures that users are given a clear message and navigation options when they reach a non-existent route.

How to Handle 404 Pages in React with react-router-dom

- First, create a new React component to represent your 404 page. This component will display a message and optionally provide navigation options.

```jsx
// src/NotFound.js

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

const NotFound = () => {
  return (
    <div style={{ textAlign: 'center', marginTop: '50px' }}>
      <h1>404 - Page Not Found</h1>
      <p>Oops! The page you're looking for doesn't exist.</p>
      <Link to="/">Go back to Home</Link>
    </div>
  );
};

export default NotFound;
```

- In your main **App.jsx** file (or wherever you have defined your routes), import the **NotFound** component and **add a route** that matches any path that isn't explicitly defined.

```jsx
// src/App.js

import React from 'react';
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';
import Home from './Home';
import About from './About';
import NotFound from './NotFound'; // Import the 404 component

const App = () => {
  return (
    <Router>
      <Routes>
        {/* Define your known routes */}
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />

        {/* Fallback route for 404 */}
        <Route path="*" element={<NotFound />} />
      </Routes>
    </Router>
  );
};

export default App;
```

- **<Route path="*">: The path="*"** route acts as a catch-all. It matches any route that is not explicitly defined above it. This is where the **NotFound** component is rendered, ensuring that any undefined path will show the 404 page.