React Router Exercise: Building a Product Management System

Welcome to our React Router exercises! In this set of exercises, you'll learn how to build a basic product management system using React Router. You'll start by setting up the routing system in the main app component, then move on to constructing shared layout components, protected routes, and child components like product details and add/edit forms. Along the way, you'll also learn how to integrate authentication and handle undefined routes. Finally, you'll enhance several existing components with features related to React Router. These exercises aim to provide hands-on experience with React Router, helping you understand how it works and apply these concepts in practical scenarios. Let's get started!

Evaluation and Testing

Objective: This section is dedicated to ensuring that all the functionalities you've implemented throughout the exercise are working correctly. Periodically evaluating and testing your work is crucial in a real-world development setting, helping you to catch and address issues before they become bigger problems.

Instructions:

1. **Checking Console for Errors**:
    - Always ensure there are no errors in the browser console. This is often the first indicator of any issues in your code.
2. **Navigation and Route Validation**:
    - Navigate through every route you've set up in the application. Make sure that each route loads the expected component and handles edge cases (like undefined routes) appropriately.
3. **SharedLayout Consistency**:
    - As you navigate through the application, verify that the shared layout elements like the header, navigation, and footer remain consistent across different pages.
4. **Protected Routes**:

- Try accessing protected routes without authentication by passing the user prop ass null. You should be redirected to the home page, signaling unauthenticated users are appropriately blocked.
- After authentication, ensure you can access these routes without any issues.

5. **Functionality of the `Product`, `AddProduct`, and `EditProduct` Components**:
    - For the `Product` component, check that it displays the right details for each product ID.
    - When adding a product using the `AddProduct` component, ensure that the product gets added and that you're navigated to the desired page afterward.
    - The `EditProduct` component should accurately fetch and display product details based on the URL's product ID. After editing, ensure the changes are saved and that navigation occurs as expected.

Remember, the objective is not just about building features but ensuring they work seamlessly and efficiently. Regular evaluation helps maintain the quality and reliability of your application. In the real world, these evaluations often lead to iterations where improvements and bug fixes are made.

Section 1: Configuring the App.jsx Component

Objective: The goal of this section is to guide you through the process of building the primary `App` component for the web application. This component will establish the necessary routing system to navigate through the various parts of the application.

Instructions:

1. **Defining the Routing System**:
    - Begin by defining the `routes` array. This array will store objects that dictate how navigation within your application operates.
    - Each route object should define a `path` that represents a URL segment and an `element` which specifies what component should be rendered when that path is accessed.
    - You can nest routes by including a `children` property in a route object. Nested routes are useful when a part of your application has a shared layout across multiple sub-pages.

2. **Setting up the Base Route**:

- Create the first object in the `routes` array. This object should represent the root path, `'/'`.
- For this route, use the `SharedLayout` component as the `element`. Pass the `user` data to it as a prop.
- As this route has several child routes (like Home, AddProduct, etc.), you will need to set up a `children` property that is an array of these child route objects.

3. **Configuring Child Routes**:

- Set up the `index` route that represents the default child of the root path. This route should render the `Home` component and receives the `products` data as props.
- Create another child route for adding a product. This path should be 'add' and it should use the `ProtectedRoute` component wrapping the `AddProduct` component. Pass the `user` prop to `ProtectedRoute`.
- Next, configure a nested route for products. It should be an object and the path should be 'products' and it should further have child routes for specific product details and editing a product. Use the dynamic `:productId` in the path to cater for any product ID.
  - For viewing a specific product, render the `Product` component with access to the `products` data.
  - For editing a product, use the `ProtectedRoute` wrapping the `EditProduct` component. Again, ensure the `ProtectedRoute` receives the `user` prop.

4. **Handling Undefined Routes**:

- To ensure user-friendly behavior, set up routes to handle undefined or erroneous paths:
  - As a catch-all for any undefined paths, add a route with the path `'*'` and have it render the `NotFound` component.

5. **Implementing the Router**:

- Now that your `routes` array is defined, create a browser router using the `createBrowserRouter` function and passing the `routes` array to it.
- Return the `RouterProvider` component. This provider requires a `router` prop which should receive the browser router you just created.

```
const App = () => {
    const router = createBrowserRouter(routes);

    return (
        <RouterProvider router={router} />
    );
};
```

Hints:

- The `ProtectedRoute` component is specially designed to restrict access to certain routes based on user authentication. It's essential to wrap components like `AddProduct` and `EditProduct` in this component to prevent unauthorized access.
- Using dynamic segments like `:productId` in your paths enables React Router to extract parameters from the URL, which can be later used to fetch specific data or perform certain actions.

Section 2: Constructing the SharedLayout Component

Objective: In this section, you will create the `SharedLayout` component. This component will act as a shell for other components, ensuring consistent elements such as the header, navigation, and footer are present across multiple pages of your web application.

Instructions:

1. **Initializing the Component**:
    - The necessary imports are given: `Outlet`, and `NavLink`.
    - Next, define the `SharedLayout` functional component that takes `user` as a prop.
2. **Structuring the Layout**:
    - Inside the component's return statement, commence by embedding a `div` element with a class of `container`. This div will serve as the main container for our layout elements.
3. **Constructing the Navigation**:

- Begin by adding a navigation link for the homepage: use a list item (`li`) encapsulating a `NavLink` component. This component, stemming from `react-router-dom`, will navigate users to the root path (`/`). To make the active link more distinguishable, utilize a callback function for its `className` prop to conditionally assign the class 'active'.

```
<li>
    <NavLink
        to="/"
        className={({ isActive }) => isActive ? 'active' : undefined}>
        Home
    </NavLink>
</li>
```

- Display an additional option allowing them to add a product. Again, wrap a `NavLink` component inside a list item (`li`). This link should navigate users to `/add`, and also make use of the conditional class assignment.

4. **Configuring the Main Content Rendering Area**:

- Below the navigation, you should structure the main content area using a `main` tag.
- The main content area is particularly special: it dynamically displays content based on the currently active route. To facilitate this, insert the `Outlet` component from `react-router-dom` inside the `main` section. Think of the `Outlet` as a dynamic placeholder; it changes the rendered component according to the path.

Section 3: Creating the ProtectedRoute Component

Objective: This section will guide you through creating a `ProtectedRoute` component, which ensures that certain routes are only accessible to authenticated users. You'll be utilizing the `useNavigate` hook from `react-router-dom` for navigation within this component.

Instructions:

1. **Using the `useNavigate` Hook**:

- Inside the `ProtectedRoute` component, initialize a variable called `navigate` using the `useNavigate` hook. This hook provides a function for programmatic navigation.

2. **Checking Authentication**:

- In the component, implement a conditional check inside a `useEffect` hook to determine if the `user` is authenticated. You can do this by checking if `user` exists. Don't forget to pass to the `useEffect` hook dependencies array the required dependencies.

```
 useEffect(() => {
    if (!user) {
        navigate("/");
    }
}, [navigate, user]);
```

- If the user is not authenticated (i.e., `!user`), you want to redirect them to an appropriate page. In this example, it assumes a route `'/'` as the redirect destination.

3. **Redirecting Unauthenticated Users**:

   - To redirect users when they are not authenticated, use the `navigate` function with the appropriate route (e.g., `navigate('/')`). This will take them to the designated destination route.

4. **Rendering Child Components for Authenticated Users**:

   - If the user is authenticated, you want to render the child components. Return `children` within curly braces `{children}`. This allows the children components passed to `ProtectedRoute` to be rendered when the user is authenticated.

Section 4: Building the `Product` Component

Objective: In this section, we'll develop a `Product` component that will display detailed information about a product based on its ID from the URL. We'll make use of the `Link` and `useParams` hooks from `react-router-dom` to achieve this.

Instructions:

1. **Extracting Product ID from the URL**:

   - Within the `Product` component, initialize a constant named `productId` by destructuring it from the `useParams()` hook. This hook extracts parameters from the current route, and in our case, we're interested in the product's ID.

2. **Identifying the Relevant Product**:

- With the `productId` at hand, our next step is to identify the corresponding product from the `products` prop.
- Declare a constant called `product` and set it to the result of the `find` method on the `products` array.
- The `find` method's callback should compare the `id` of each product to the `productId`. Remember to convert `productId` to an integer using `parseInt` since parameters fetched using `useParams` are strings by default.

3. **Handling Product Absence**:

   - Considering that the product may not always exist (e.g., if an incorrect ID is manually input into the URL), we need to handle such scenarios gracefully.
   - Implement a conditional check: If the `product` constant is undefined, return a `div` element with the text "Product not found!".

4. **Facilitating Product Edits**:

   - Equip users with the capability to modify the product details.
   - Implement the `Link` component from `react-router-dom` to navigate to the product's edit page.
   - Set the `to` prop of the `Link` component to a dynamic value: `/products/${productId}/edit`.
   - Give the `edit-link` class to this link for styling and label the link "Edit Product".

Section 5: Building the `AddProduct` Component with Navigation

Objective: The purpose of this section is to guide you through enhancing the `AddProduct` component by adding navigation capabilities. This ensures that after a product is added, the user is navigated back to the home page or another appropriate page.

Instructions:

1. **Setting Up Navigation**:

   - Invoke the `useNavigate` hook and assign the returned function to a constant named `navigate`. This function will be used to programmatically navigate the user.

2. **Constructing the `handleSubmit` Function with Navigation**:

- After logging the product and resetting the state, employ the `navigate` function with `'/'` as its argument. This action will redirect the user to the home page after they've added a product.

Section 6: Enhancing the `EditProduct` Component with React Router

Objective: The purpose of this section is to enhance the `EditProduct` component by integrating functionalities associated with React Router. By the end, you'll have a component that fetches the product to edit based on the ID from the URL and navigates back to the home page after the editing is done.

Instructions:

1. **Retrieving the Product ID from the URL**:
   - Using the `useParams` hook from `react-router-dom`, extract the `productId` parameter from the current URL.
2. **Setting Up Programmatic Navigation**:
   - Invoke the `useNavigate` hook from `react-router-dom` and assign the returned function to a constant named `navigate`. This function will allow you to programmatically redirect users after the product edits have been saved.
3. **Fetching the Product Data Based on URL Parameter**:
   - With the `productId` extracted in step 1, search for the corresponding product within the `products` mock data. Ensure you parse the `productId` to an integer for accurate comparison.
4. **Initializing the Form Data with the Fetched Product's Information**:
   - Set the initial state of the `formData` using the `useState` hook. Populate it with the properties of the found product (i.e., `name`, `description`, `price`, and `stock`).

```
const [formData, setFormData] = useState({
    name: product.name,
    description: product.description,
    price: product.price,
    stock: product.stock,
});
```

5. **Enhancing the `handleSubmit` Function with Navigation**:
    - After handling the form's submit action (like saving edits), employ the `navigate` function with `'/'` as its argument. This step will ensure the user gets redirected to the home page after the edits have been submitted.

```
const handleSubmit = (e) => {
    e.preventDefault();
    navigate(`/`);
};
```

Conclusion

Congratulations on completing this comprehensive React Router tutorial! Throughout this exercise, you learned how to build an effective routing solution using React Router. You covered topics ranging from configuring base routes, constructing child routes, handling undefined routes, setting up protected routes, implementing react router, building layout components, working with Link components, and much more. These skills will undoubtedly help you in your career as a developer and enable you to build robust, modern web applications that meet today's demands.