# React Router 6: Private Routes (alias Protected Routes)

FEBRUARY 06, 2022 BY ROBIN WIERUCH - EDIT THIS POST



Private Routes in React Router (also called **Protected Routes**) require a user being authorized to visit a route (read: page). So if a user is not authorized for a specific page, they cannot access it. The most common example is authentication in a React application where a user can only access the protected pages when they are authorized (which means in this case being authenticated). Authorization goes beyond authentication though. For example, a user can also have roles and permissions which give a user access to specific areas of the application.

This is a React Router tutorial which teaches you how to use **Private Routes with React Router 6**. The code for this React Router v6 tutorial can be found over here.

Continue Reading: React Router 6 Introduction

We will start off with a minimal React project that uses React Router to navigate a user from one page to another page. In the following function component, we have matching Link and Route components from React Router for various routes. Furthermore, we have a so-called Index Route loaded with the Landing component and a so-called No Match Route loaded with inline JSX. Both act as fallback routes:

```
import { Routes, Route, Link } from 'react-router-dom';

const App = () => {
  return (
    <>
      <h1>React Router</h1>

      <Navigation />

      <Routes>
        <Route index element={<Landing />} />
        <Route path="landing" element={<Landing />} />
        <Route path="home" element={<Home />} />
        <Route path="dashboard" element={<Dashboard />} />
        <Route path="analytics" element={<Analytics />} />
        <Route path="admin" element={<Admin />} />
        <Route path="*" element={<p>There's nothing here: 404!</p>} />
      </Routes>
    </>
  );
};

const Navigation = () => (
  <nav>
    <Link to="/landing">Landing</Link>
    <Link to="/home">Home</Link>
    <Link to="/dashboard">Dashboard</Link>
    <Link to="/analytics">Analytics</Link>
    <Link to="/admin">Admin</Link>
  </nav>
);
```

In the following, we want to protect all routes (except for the Landing route, because it's a public route) from unauthorized access. Each page has a different authorization mechanism. Only the Home and Dashboard pages share the same authorization requirements:

```
const Landing = () => {
  return <h2>Landing (Public: anyone can access this page)</h2>;
};

const Home = () => {
  return <h2>Home (Protected: authenticated user required)</h2>;
};

const Dashboard = () => {
  return <h2>Dashboard (Protected: authenticated user required)</h2>;
};

const Analytics = () => {
  return (
    <h2>
      Analytics (Protected: authenticated user with permission
      'analyze' required)
    </h2>
  );
};

const Admin = () => {
  return (
    <h2>
      Admin (Protected: authenticated user with role 'admin' required
    </h2>
  );
};
```

We will start off by simulating a user login/logout mechanism. By using two buttons conditionally rendered, we either render a login or logout button based on the authentication status of the user. Based on the event handler, we either set a user or reset it to null by using React's useState Hook:

```
const App = () => {
  const [user, setUser] = React.useState(null);

  const handleLogin = () => setUser({ id: '1', name: 'robin' });
  const handleLogout = () => setUser(null);

  return (
    <>
      <h1>React Router</h1>

      <Navigation />

      {user ? (
        <button onClick={handleLogout}>Sign Out</button>
      ) : (
        <button onClick={handleLogin}>Sign In</button>
      )}

      <Routes>
        <Route index element={<Landing />} />
        <Route path="landing" element={<Landing />} />
        <Route path="home" element={<Home user={user} />} />
        ...
      </Routes>
    </>
  );
};
```

The user will serve us either as logged in or logged out user. Next we are going to protect our first route. Therefore, we will start by implementing a redirect with React Router in the Home component where we already passed the `user` as prop to the component:

```
import { Routes, Route, Link, Navigate } from 'react-router-dom';

...

const Home = ({ user }) => {
  if (!user) {
    return <Navigate to="/landing" replace />;
  }

  return <h2>Home (Protected: authenticated user required)</h2>;
};
```

When there is a logged in user, the Home component does not run into the if-else condition's block and renders the actual content of the Home component instead. However, if there is no logged in user, the Home component renders React Router's Navigate component and therefore redirects a user to the Landing page. In the case of a user being on the Home page and logging out by clicking the button, the user will experience a redirect from the protected page.

Continue Reading: React Router 6 Redirect

We protected our first React component with React Router. However, this approach does not scale, because we would have to implement the same logic in every protected route. In addition, the redirect logic should not reside in the Home component itself but as a best practice protect it from the outside instead. Therefore, we will extract the logic into a standalone component:

```
const ProtectedRoute = ({ user, children }) => {
  if (!user) {
    return <Navigate to="/landing" replace />;
  }

  return children;
};
```

Then we can use this new protecting route component as wrapper for the Home component. The Home component itself does not need to know about this guarding mechanism anymore:

```
const App = () => {
  ...

  return (
    <>
      ...

      <Routes>
        <Route index element={<Landing />} />
        <Route path="landing" element={<Landing />} />
        <Route
          path="home"
          element={
            <ProtectedRoute user={user}>
              <Home />
            </ProtectedRoute>
          }
        />
        ...
      </Routes>
    </>
  );
};

const Home = () => {
  return <h2>Home (Protected: authenticated user required)</h2>;
};
```

This new protecting route component acts as abstraction layer for the whole authorization mechanism to protect certain pages from unauthorized access. Because we extracted it as reusable component, which can be used to compose another component (or components) into it, we can extend the implementation details too. For example, in most cases (here: a user not being authenticated) we want to redirect a user to a public route (e.g. `'/landing'`). However, we can also be specific about the redirected path by

using an optional prop:

```
const ProtectedRoute = ({
  user,
  redirectPath = '/landing',
  children,
}) => {
  if (!user) {
    return <Navigate to={redirectPath} replace />;
  }

  return children;
};
```

We will come back to extending this component when we have to deal with permissions and roles. For now, we will reuse this component for other routes which need the same level of protection. For example, the Dashboard page requires a user to be logged in too, so let's protect this route:

```
const App = () => {
  ...

  return (
    <>
      ...

      <Routes>
        <Route index element={<Landing />} />
        <Route path="landing" element={<Landing />} />
        <Route
          path="home"
          element={
            <ProtectedRoute user={user}>
              <Home />
            </ProtectedRoute>
          }
        />
        <Route
          path="dashboard"
          element={
            <ProtectedRoute user={user}>
              <Dashboard />
            </ProtectedRoute>
          }
        />
        <Route path="analytics" element={<Analytics />} />
        <Route path="admin" element={<Admin />} />
        <Route path="*" element={<p>There's nothing here: 404!</p>} /
      </Routes>
    </>
  );
};
```

A better way of protecting both sibling routes with the same authorization level would be using a Layout Route wich renders the ProtectedRoute component for both nested routes:

```
import {
  Routes,
  Route,
  Link,
  Navigate,
  Outlet,
} from 'react-router-dom';

const ProtectedRoute = ({ user, redirectPath = '/landing' }) => {
  if (!user) {
    return <Navigate to={redirectPath} replace />;
  }

  return <Outlet />;
};

const App = () => {
  ...

  return (
    <>
      ...

      <Routes>
        <Route index element={<Landing />} />
        <Route path="landing" element={<Landing />} />
        <Route element={<ProtectedRoute user={user} />}>
          <Route path="home" element={<Home />} />
          <Route path="dashboard" element={<Dashboard />} />
        </Route>
        <Route path="analytics" element={<Analytics />} />
        <Route path="admin" element={<Admin />} />
        <Route path="*" element={<p>There's nothing here: 404!</p>} /
      </Routes>
    </>
  );
};
```

By using React Router's Outlet component instead of React's children prop, you can use the ProtectedRoute component as Layout component. However, when attempting to use the ProtectedRoute as wrapping component as before your application will break. Therefore, you can optionally render the children when the ProtectedRoute is not used as Layout component:

```
const ProtectedRoute = ({
  user,
  redirectPath = '/landing',
  children,
}) => {
  if (!user) {
    return <Navigate to={redirectPath} replace />;
  }
```

```
  return children ? children : <Outlet />;
};
```

That's it for the essential protection of private routes which covers the essential case of having an authenticated user. However, in a more complex application you will encounter permissions and roles too. We will simulate both cases by giving our user a permission and role in arrays, because they could have multiple of them:

```
const App = () => {
  const [user, setUser] = React.useState(null);

  const handleLogin = () =>
    setUser({
      id: '1',
      name: 'robin',
      permissions: ['analyze'],
      roles: ['admin'],
    });

  const handleLogout = () => setUser(null);

  return (...);
};
```

So far, the ProtectedRoute component only deals with authenticated users as authorization process. We need to extend it to handle permissions and roles too. Therefore, we will enable developers to pass in a boolean as condition which acts as more abstract guard for rendering the protected component:

```
const ProtectedRoute = ({
  isAllowed,
  redirectPath = '/landing',
  children,
}) => {
  if (!isAllowed) {
    return <Navigate to={redirectPath} replace />;
  }

  return children ? children : <Outlet />;
};
```

Because we defined this condition previously in the ProtectedRoute component itself, we need to define the condition from the outside now. This applies to our so far protected routes in addition to the new protected routes which require the user to have a certain permission or role:

```
const App = () => {
  ...

  return (
    <>
      ...

      <Routes>
        <Route index element={<Landing />} />
        <Route path="landing" element={<Landing />} />
        <Route element={<ProtectedRoute isAllowed={!!user} />}>
          <Route path="home" element={<Home />} />
          <Route path="dashboard" element={<Dashboard />} />
        </Route>
        <Route
          path="analytics"
          element={
            <ProtectedRoute
              redirectPath="/home"
              isAllowed={
                !!user && user.permissions.includes('analyze')
              }
            >
              <Analytics />
            </ProtectedRoute>
          }
        />
        <Route
          path="admin"
          element={
            <ProtectedRoute
              redirectPath="/home"
              isAllowed={!!user && user.roles.includes('admin')}
            >
              <Admin />
            </ProtectedRoute>
          }
        />
        <Route path="*" element={<p>There's nothing here: 404!</p>} />
      </Routes>
    </>
  );
};
```

While the Home and Dashboard pages require a user to be present (read: authenticated), the Analytics and Admin pages require a user to be authenticated and to have certain permissions/roles. Try it yourself by revoking the user either their roles or permissions.

Furthermore, the protected route for the Analytics and Admin pages makes use of the optional `redirectPath`. If a user does not fulfil the permissions or roles authorization requirements, the user gets redirected to the protected Home page. If there is a user that's not authenticated in the first place, they get redirected to the Landing page.

Continue Reading: React Router 6 Authentication

If you are a fan of Higher-Order Components, you could created a protected route with HoCs as well. Anyway, I hope this tutorial helped you to understand private routes (alias protected routes) in React Router and how to use them as guards for routes that require

authorization based on the authentication status of a user or their roles and permissions.

Discuss on Twitter Share on Twitter

## KEEP READING ABOUT REACT>

### REACT FIREBASE AUTHORIZATION WITH ROLES

So far, you've used broad authorization rules that check user authentication, where the dedicated authorization higher-order component redirects them to the login page if the user is not authenticated...

### REACT ROUTER 6: NESTED ROUTES

A React Router tutorial which teaches you how to use Nested Routes with React Router 6 . The code for this React Router v6 tutorial can be found over here . In order to get you started, create a new...