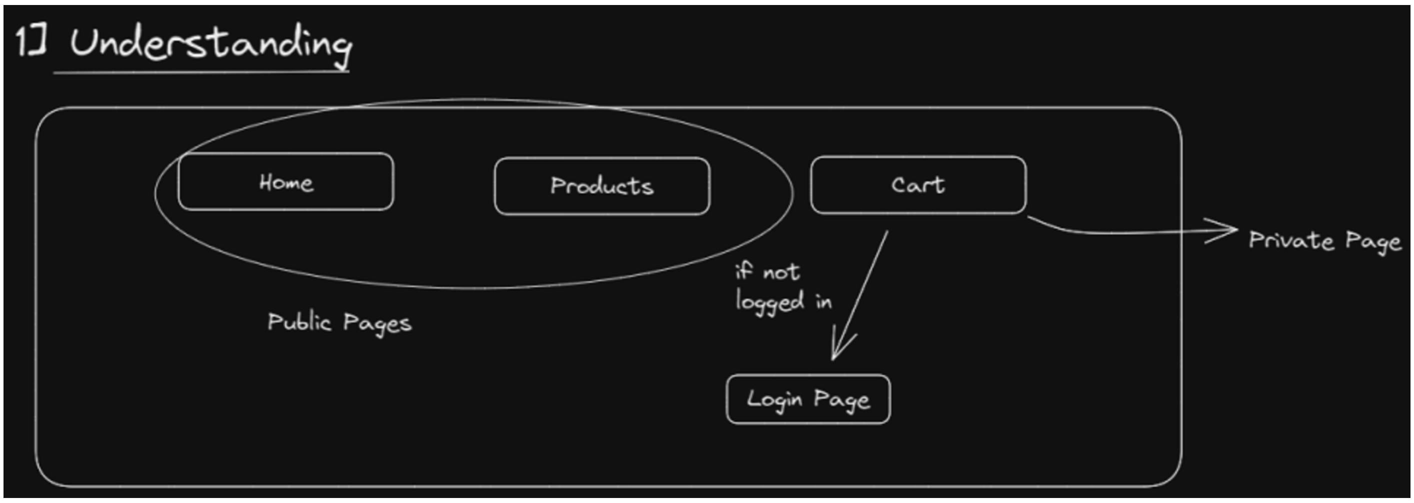
Frontend Authentication Notes

Why do we need front-end authentication?

- Protect the private routes/pages example wishlist page, profile page, and details page.
- To protect the private APIs.
- If the user is not logged in, redirect to the login page.

Pictorial explanation -



1. Basic way to create frontend authentication -

1. Create a component with the name `Address.js` that will render the address details.
2. Create another component with the name `Login.js` that will render the login component.
3. Add the routing for the Address component in App.js for the path `/address`.
4. Create a variable with the name `isLoggedIn` which will contain a boolean value with an initial value of false.
5. Create a login button in App.js which will toggle the state variable `isLoggedIn` between true and false.
6. If `isLoggedIn` is true then render the Address component otherwise if `isLoggedIn` is false then render the Login component.

Address.js

```
export const Address = () => {
  return <h1>This is address</h1>;
};
```

COPY

Login.js

```
export const Login = () => {
  return <h1>Please Login</h1>;
};
```

COPY

App.js

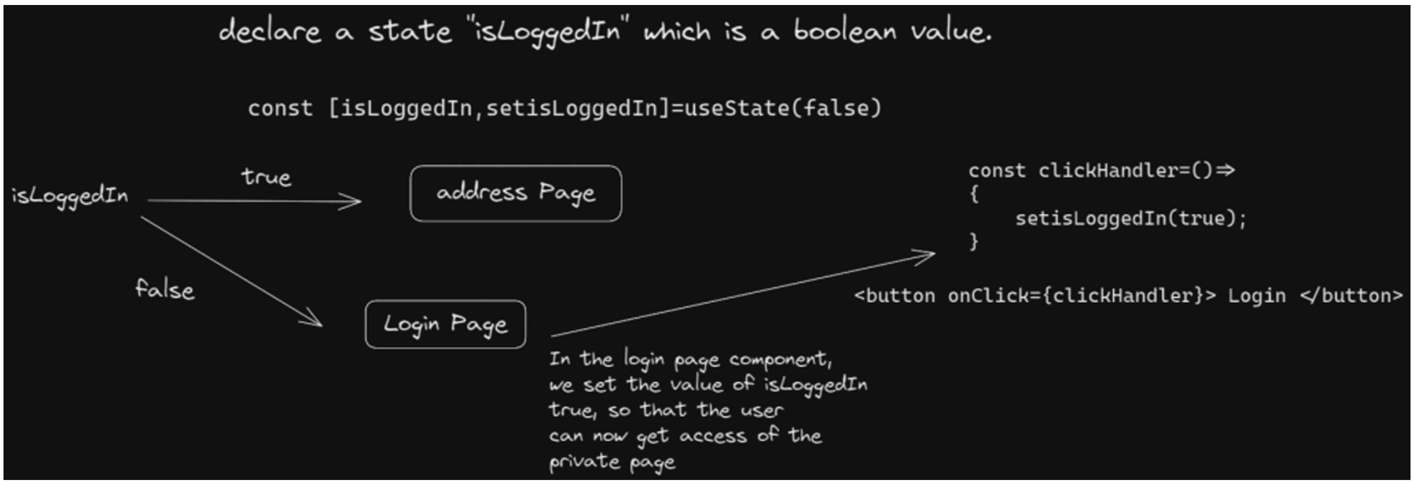
```
export default function App() {
  const [isLoggedIn, setisLoggedIn] = useState(false);
  return (
    <div className="App">
      <nav>
        <NavLink style={getActiveStyle} to="/">
          Home
        </NavLink>
        ||
        <NavLink style={getActiveStyle} to="/address">
          Address
        </NavLink>
      </nav>

      <button onClick={() => setisLoggedIn(!isLoggedIn)}>
        {isLoggedIn ? "Logout" : "Login"}
      </button>

      <Routes>
        <Route path="/" element={<Home />} />
        {isLoggedIn && <Route path="/address" element={<Address />} />}
        {!isLoggedIn && <Route path="/address" element={<Login />} />}
      </Routes>
    </div>
  );
}
```

COPY

Pictorial explanation -



declare a state "isLoggedIn" which is a boolean value.

```
const [isLoggedIn,setisLoggedIn]=useState(false)
```

```
const clickHandler=()=>
{
    setisLoggedIn(true);
}
```

```
<button onClick={clickHandler}> Login </button>
```

In the login page component,
we set the value of isLoggedIn
true, so that the user
can now get access of the
private page

Cons of the above method -

- No reusability of the `isLoggedIn` state variable.
- Multiple routing conditions based upon the value of `isLoggedIn` which is making the code messy.
- If a login has to be implemented in other child components then `isLoggedIn` has to be passed as a prop.
- To tackle this problem, we will create a reusable authentication component.

2. Creating frontend authentication with a separate component

1. Create a separate component name `RequiresAuth.js` that will take 2 parameters, one is the boolean state variable `isLoggedIn` and the other is the children component.
2. Inside the `RequiresAuth.js`, if the `isLoggedIn` is true then render the children component otherwise with the help of `<Navigate>`, render the component that is on the path `/login` which is the `Login.js` component.
3. In App.js, create route for /login and render`Login.js` for this route.
4. In App.js, import the `RequiresAuth.js` component.
5. Wrap the components that have to be protected (in our case `Address.js`) with `RequiresAuth.js` and pass the prop `isLoggedIn` to the component.

RequiresAuth.js

```
import { Navigate } from "react-router-dom";

export default function RequiresAuth({ isLoggedIn, children }) {
  return isLoggedIn ? children : <Navigate to="/login" />;
}
```

COPY

App.js

```
export default function App() {
  const [isLoggedIn, setIsLoggedIn] = useState(true);
  return (
    <div className="App">
      <nav>
        <NavLink style={getActiveStyle} to="/">
          Home
        </NavLink>
        ||
        <NavLink style={getActiveStyle} to="/address">
          Address
        </NavLink>
        <button onClick={() => setIsLoggedIn(!isLoggedIn)}>
          {isLoggedIn ? "Logout" : "Login"}
        </button>
      </nav>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/login" element={<Login />} />
        <Route
          path="/address"
          element={
            <RequiresAuth isLoggedIn={isLoggedIn}>
              <Address />
            </RequiresAuth>
          }
        />
      </Routes>
    </div>
  );
}
```

COPY

Cons of the above method -

- No reusability of the `isLoggedIn` state variable.
- If a login has to be implemented in other child components then `isLoggedIn` has to be passed as a prop.
- Pass `isLoggedIn` to `RequiresAuth` component every time we need to implement protected routes.
- To tackle this situation we will make use of `Context`.

3) Create frontend authentication with the help of useContext

1. Create an `AuthContext.js` file, within the file, create `AuthContext` with the help of `createContext()`.
2. Create a variable with the name `isLoggedIn` which will contain a boolean value with an initial value of false.
3. Provide the `isLoggedIn` state variable and `setIsLoggedIn` function to the children with the help of `<AuthContext.Provider>` within `AuthProvider` function.
4. Import the `AuthProvider` inside index.js and wrap `App.js` with `AuthProvider` so that context is available inside `App.js`.
5. Now inside App.js, consume the context with the help of `useContext()` and destructure `isLoggedIn` and `setIsLoggedIn`.
6. With the help of a button, toggle the value of `isLoggedIn` with the help of `setIsLoggedIn`.
7. Inside the `RequiresAuth.js`, if the `isLoggedIn` is true then render the children component otherwise with the help of `<Navigate>`, render the component that is on the path `/login` which is the `Login.js` component.

AuthContext.js

```
import { createContext, useState } from "react";
export const AuthContext = createContext({ isLoggedIn: false });
export function AuthProvider({ children }) {
  const [isLoggedIn, setIsLoggedIn] = useState(false);

  return (
    <AuthContext.Provider value={{ isLoggedIn, setIsLoggedIn }}>
      {children}
    </AuthContext.Provider>
  );
}
```

COPY

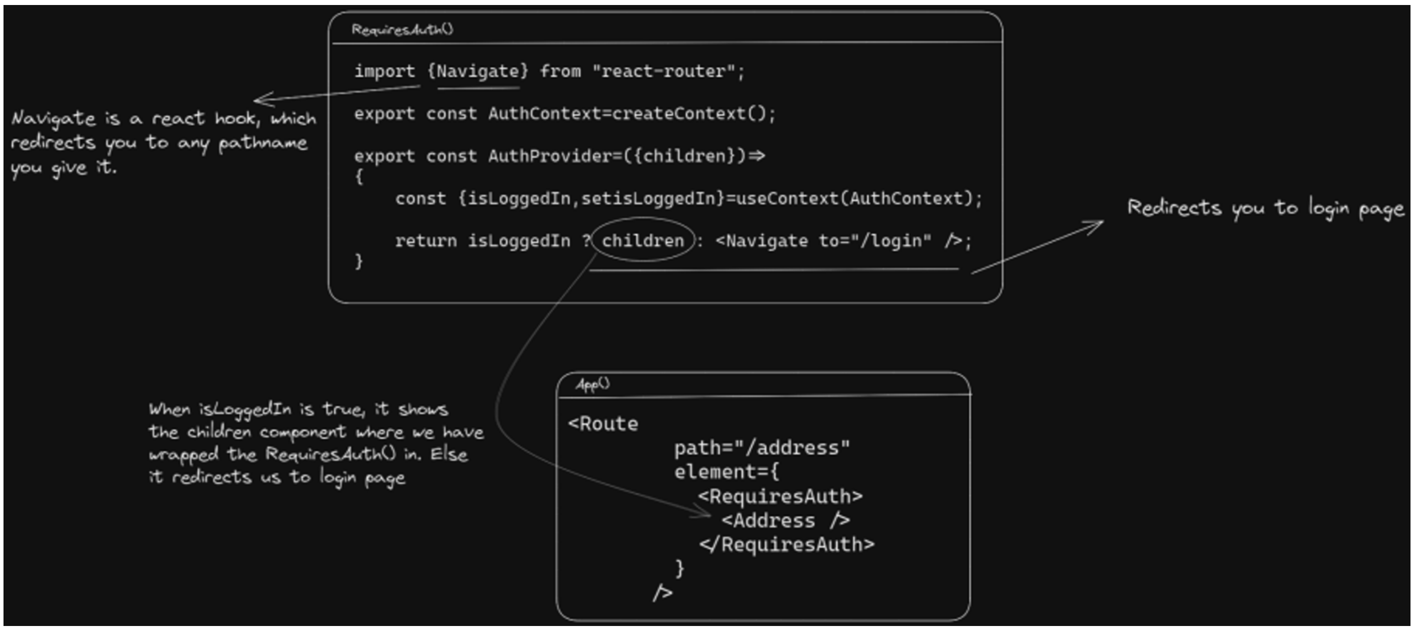
RequiresAuth.js

```
import { useContext } from "react";
import { Navigate, useLocation } from "react-router";
import { AuthContext } from "..";

export function RequiresAuth({ children }) {
  const { isLoggedIn } = use context(AuthContext);
  return isLoggedIn ? (
    children
  ) : (
    <Navigate to="/login" />
  );
}
```

COPY

Pictorial explanation -



```
RequiresAuth()

import {Navigate} from "react-router";

export const AuthContext=createContext();

export const AuthProvider=({children})=>
{
    const {isLoggedIn,setisLoggedIn}=useContext(AuthContext);

    return isLoggedIn ? children : <Navigate to="/login" />;
}
```

```
App()

<Route
         path="/address"
         element={
           <RequiresAuth>
             <Address />
           </RequiresAuth>
         }
      />
```

Navigate is a react hook, which redirects you to any pathname you give it.
Redirects you to login page
When isLoggedIn is true, it shows the children component where we have wrapped the RequiresAuth() in. Else it redirects us to login page

4) How to redirect to the same URL after successful login?

What does the useLocation() hook do?

It gives the current location. Suppose we are on the login page, so the `useLocation()` will indicate the current location as `/login`.

```
// Suppose we clicked the address route with the path ("/address")
```

```
let location = useLocation();
console.log(location.pathname) // "/address"
```

COPY

What does the useNavigate() hook do?

`useNavigate()` is a hook provided by react-router. The `useNavigate()` hook returns a function to which we can pass a pathname to which we want to get redirected to.

For example, Suppose we want to get redirected to the address page -

```
const navigate = useNavigate();
navigate("/address");
```

COPY

Steps to redirect to the same URL after successful login -

1. Inside `RequiresAuth.js`, with the help of `useLocation()`, we will store the current route.
2. If the `isLoggedIn` is true (user is logged in), render the children component.
3. If the `isLoggedIn` is false (the user is not logged in), then append the current route to the `state` object of the `location` object.
4. Inside `App.js`, we will access the `location` object with the help of `useLocation()`.
5. Now, with the click of the login button, with the help of `useNavigate()` we will pass the pathname of the `state` object inside the global `location` object to `navigate` function that will help in redirection after successful login.

RequiresAuth.js

```
import { useContext } from "react";
import { Navigate, useLocation } from "react-router";
import { AuthContext } from "..";

export function RequiresAuth({ children }) {
  let location = useLocation();
  const { isLoggedIn } = useContext(AuthContext);
  return isLoggedIn ? (
    children
  ) : (
    <Navigate to="/login" state={{ from: location }} />
  );
}
```

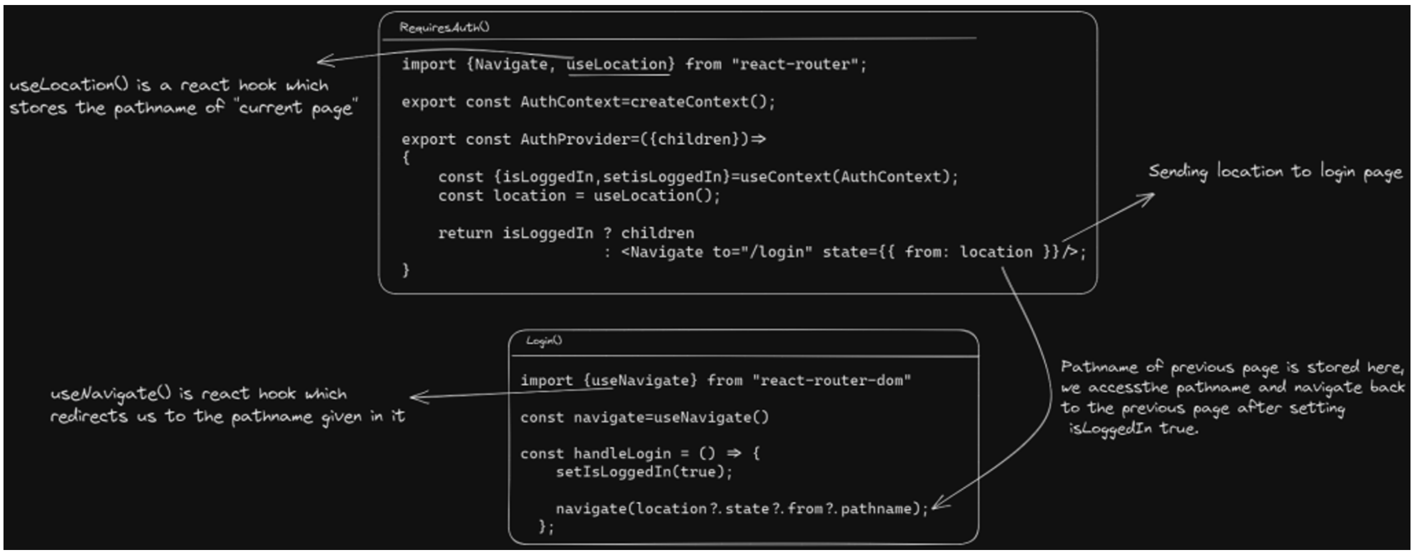
COPY

App.js

```
export default function App() {
  const { isLoggedIn, setIsLoggedIn } = useContext(AuthContext);
  const navigate = useNavigate();
  const location = useLocation();
  const handleLogin = () => {
    setIsLoggedIn(!isLoggedIn);
    navigate(location?.state?.from?.pathname);
  };
  return (
    <div className="App">
      <nav>
        <NavLink style={getActiveStyle} to="/">
          Home
        </NavLink>
        ||
        <NavLink style={getActiveStyle} to="/address">
          Address
        </NavLink>
        ||{" "}
        <button onClick={handleLogin}>{isLoggedIn ? "Logout" : "Login"}</button>
      </nav>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route
          path="/address"
          element={
            <RequiresAuth>
              <Address />
            </RequiresAuth>
          }
        />
      </Routes>
    </div>
  );
}
```

COPY

Pictorial Explanation-



useLocation() is a react hook which stores the pathname of "current page"

```
import {Navigate, useLocation} from "react-router";

export const AuthContext=createContext();

export const AuthProvider=({children})=>
{
    const {isLoggedIn,setisLoggedIn}=useContext(AuthContext);
    const location = useLocation();

    return isLoggedIn ? children
                     : <Navigate to="/login" state={{ from: location }}/>;
}
```

Sending location to login page

useNavigate() is react hook which redirects us to the pathname given in it

```
import {useNavigate} from "react-router-dom"

const navigate=useNavigate()

const handleLogin = () => {
    setIsLoggedIn(true);

    navigate(location?.state?.from?.pathname);
  };
```

Pathname of previous page is stored here, we accessthe pathname and navigate back to the previous page after setting isLoggedIn true.

Explanation -

Let's understand the mechanism in 2 parts -

When a user clicks the Address navlink (the user is not logged in)

- Since the user is not logged in at this time, `isLoggedIn` will be false and we will get redirected to the `/login` route.
- Simultaneously, during this time `useLocation()` hook will remember the current route from where we got redirected to `/login` that is `/address`.
- Now since `isLoggedIn` is false, `<Navigate>` will be called and it will redirect to the `login component` at the same time within the `location object`, further within the state object, the URL that was remembered by `useLocation()` gets appended that is `/address`.
- Remember, the user is still not logged in yet.

When a user clicks the login button after redirection to the login page

- When the user clicked the login button,`handleLogin` gets called and `isLoggedIn` turns to true.
- Now we will get the `location object` with the help of `useLocation()`.
- Now with the help of `useNavigate()`, we will pass the pathname of the previous route from where we got redirected to the login page that we appended earlier inside `location object`.
- With the successful login, we will get redirected back to the route from where we came.