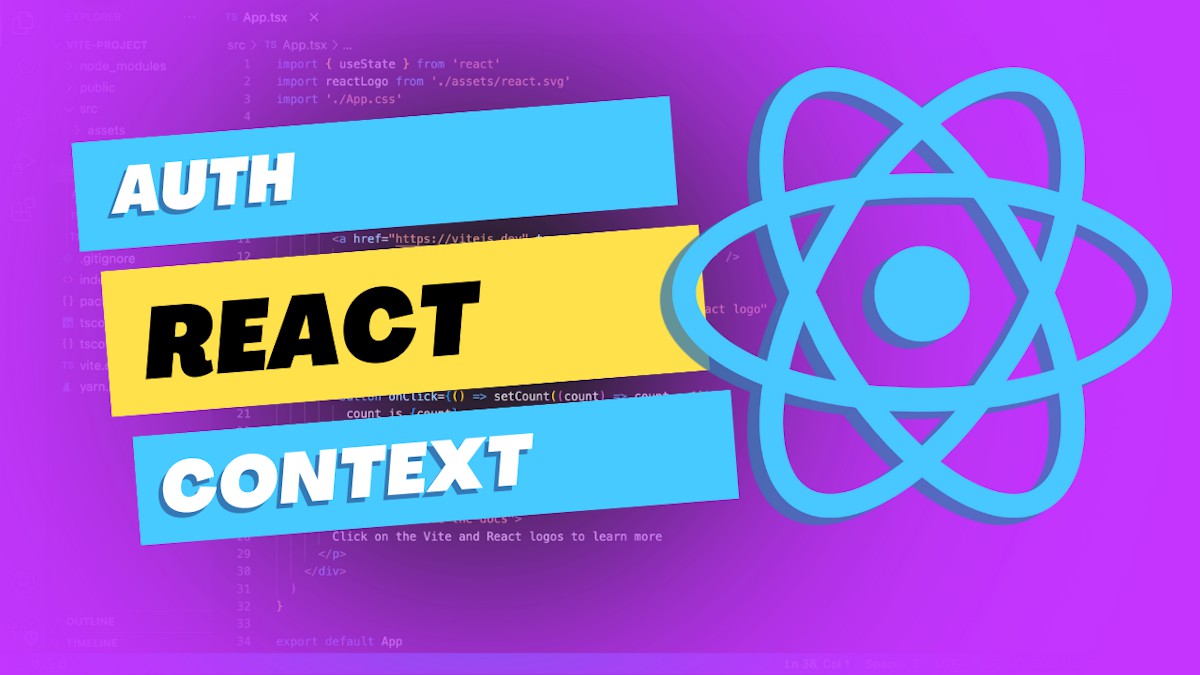
Auth & Context in React 18

[**sammeechward.com**/use-context-auth](https://www.sammeechward.com/use-context-auth)



This article is part of the following playlists:

[All You Need To Know About React.js](https://www.sammeechward.com/use-context-auth?playlist=all-you-need-to-know-about-react-js)

[Learn everything you need to know to get started building React apps.](https://www.sammeechward.com/use-context-auth?playlist=all-you-need-to-know-about-react-js)

Imagine displaying the user's name in the header, showcasing their posts on the home screen, and revealing their profile picture in the sidebar—all while keeping certain pages off-limits to unauthenticated users.

In this guide, you'll learn how to share user data between components with the context API and the useContext hook. We'll also dive into integrating this powerful technique with **react-router** to protect routes and update the nav bar based on the user's logged-in status.

# Setting Up the Project with Vite

Let's dive right into creating a new React 18 project using Vite. The goal here is to create an app that can handle user authentication using useContext. So, fasten your seat belts and let's get started!

Create a new Vite project using the React template.

npx create-vite my-auth-app --template react cd my-auth-app

npm install

Now that our project is ready, let's add the necessary components and set up our authentication context.

# Creating the Authentication Context

First things first, let's create a context that will manage the user's logged-in status and store their JWT token. We'll call it AuthContext.

Note

Using Context allows us to keep track of information with useState or useReducer, then any component can access that information without having to pass it down as props. If you're completely unfamiliar with how userContext works, check out the [official documentation](https://react.dev/reference/react/useContext).

Create a new file src/AuthContext.jsx and add the following content:

import { createContext, useState } from "react"; const AuthContext = createContext();

function AuthProvider(props) {

const [isLoggedIn, setIsLoggedIn] = useState(false); const [token, setToken] = useState(null);

const login = (jwtToken) => { setIsLoggedIn(true); setToken(jwtToken);

};

const logout = () => { setIsLoggedIn(false); setToken(null);

};

const value = { isLoggedIn, token, login, logout,

};

return <AuthContext.Provider value={value} {...props} />;

}

export { AuthContext, AuthProvider };

Here's what's happening in the code above:

1. Import createContext and useState from React.
2. Create a new context called AuthContext.
3. Define an AuthProvider function component that will wrap our entire app. This component maintains the isLoggedIn and token state.
4. Define the login and logout functions to manage the user's logged-in status and token.
5. Pass the context value as an object containing the state variables and functions.
6. Finally, export the AuthContext and AuthProvider so we can use them in other parts of our app.

Now we can make it so that any component is able to access the

isLoggedIn,token,login, and logout values.

# Creating the Login Screen

We're going to create a very simple login screen that mocks a network request for a JWT token.

Create a new file src/Login.jsx and add the following content:

import { useContext, useState } from "react"; import { AuthContext } from "./AuthContext";

function Login() {

const [email, setEmail] = useState(""); const [password, setPassword] = useState(""); const { login } = useContext(AuthContext);

const handleSubmit = async (event) => { event.preventDefault();

// Fake network request for JWT token const jwtToken = "fake-jwt-token"; login(jwtToken);

};

return (

<div>

<h1>Login</h1>

<form onSubmit={handleSubmit}>

<label htmlFor="email">Email:</label>

<input

type="email" id="email" value={email}

onChange={(e) => setEmail(e.target.value)}

/>

<br />

<label htmlFor="password">Password:</label>

<input

type="password" id="password" value={password}

onChange={(e) => setPassword(e.target.value)}

/>

<br />

<button type="submit">Login</button>

</form>

</div>

);

}

export default Login;

In the Login component above, we:

1. Import the necessary hooks and AuthContext.
2. Use the useContext hook to access the login function from our

AuthContext.

1. Define local state for email and password input fields.
2. Define a handleSubmit function that simulates a network request for a JWT token and calls the login function with the fake token.
3. Render a simple form with email and password input fields and a submit button.

# Creating the Home Screen

Now let's create a basic home screen that will display the user's logged-in status and allow them to log out.

Create a new file src/Home.jsx and add the following content:

import { useContext } from "react";

import { AuthContext } from "./AuthContext";

function Home() {

const { isLoggedIn, logout } = useContext(AuthContext);

return (

<div>

<h1>Home</h1>

{isLoggedIn ? (

<>

<p>Welcome! You are logged in.</p>

<button onClick={logout}>Logout</button>

</>

) : (

<p>Please log in to access more features.</p>

)}

</div>

);

}

export default Home;

In the Home component above, we:

1. Import the necessary hooks and AuthContext.
2. Use the useContext hook to access the isLoggedIn and logout

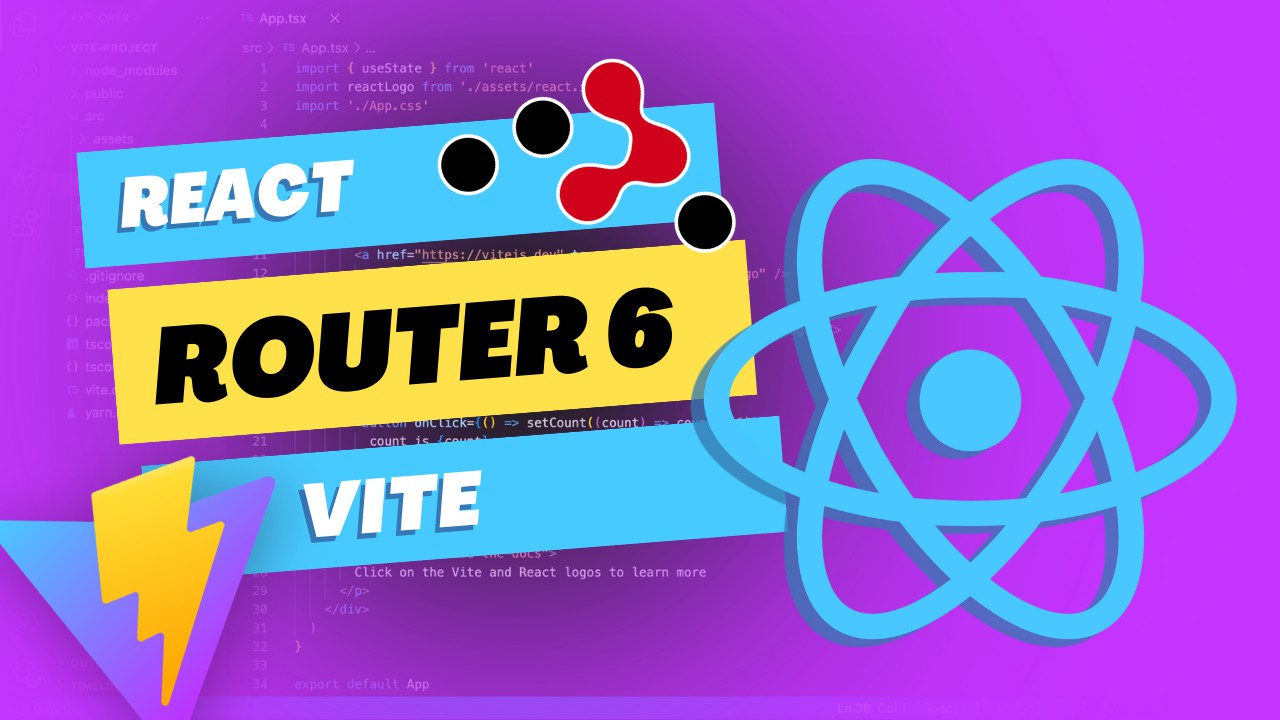
functions from our AuthContext.

1. Render a welcome message and a logout button if the user is logged in, otherwise, display a message prompting the user to log in.

# Setting Up Routes with react-router-dom

Before diving into wrapping our app with AuthProvider, let's first install react- router-dom and set up routes for our Home and Login components.

If you're not familiar with react-router-dom, check out:

[](https://www.sammeechward.com/react-router-6)

[Adding React Router 6](https://www.sammeechward.com/react-router-6)

[In this tutorial, we'll explore the process of adding React Router 6 to a React 18 application using...](https://www.sammeechward.com/react-router-6)

[](https://www.sammeechward.com/react-router-6)

[1 year ago](https://www.sammeechward.com/react-router-6)

Install React Router 6 as a dependency.

npm install react-router-dom

Now, modify src/App.jsx to include the new routes:

import { BrowserRouter, Link, Route, Routes } from "react-router-dom"; import Home from "./Home";

import Login from "./Login";

function App() { return (

<BrowserRouter>

<nav>

<ul>

<li>

<Link to="/">Home</Link>

</li>

<li>

<Link to="/login">Login</Link>

</li>

</ul>

</nav>

<main>

<Routes>

<Route path="/" element={<Home />} />

<Route path="/login" element={<Login />} />

</Routes>

</main>

</BrowserRouter>

);

}

export default App;

So we have a login screen and a home screen, but if you try running the app right now, it won't work. That's because we haven't wrapped our app with AuthProvider yet. A component must be nested inside AuthProvider in order to access the context values.

# Wrapping the App with AuthProvider

With our routes in place, let's wrap our entire app with the AuthProvider. This will make the user data available to any component nested inside it.

Modify src/App.jsx to include the AuthProvider:

import { BrowserRouter, Link, Route, Routes } from "react-router-dom"; import { AuthProvider } from "./AuthContext";

import Home from "./Home"; import Login from "./Login";

function App() { return (

<AuthProvider>

<BrowserRouter>

<nav>

<ul>

<li>

<Link to="/">Home</Link>

</li>

<li>

<Link to="/login">Login</Link>

</li>

</ul>

</nav>

<main>

<Routes>

<Route path="/" element={<Home />} />

<Route path="/login" element={<Login />} />

</Routes>

</main>

</BrowserRouter>

</AuthProvider>

);

}

export default App;

# Testing the App

Now that everything is set up, it's time to test our app.

Start the development server.

npm run dev

Open your browser and navigate to http://localhost:5173. You should see the home page displaying the "Please log in to access more features." message. Click on the "Login" link in the navigation menu and enter any email and password to log in. After logging in, you can navigate back to the home page, which will now display a welcome message and a logout button.

And that's it! We've successfully created an app that uses React 18's useContext hook to manage user authentication. Remember, this tutorial uses a fake JWT token for simplicity, but in a real-world scenario, you'd want to replace it with a proper network request to your authentication server.

# Redirect to Home After Login

After the user logs in, they should be navigated to a different page. In this case, we'll redirect them to the home page.

Modify src/Login.jsx to redirect the user to the home page after logging in:

import { useContext, useState } from "react"; import { AuthContext } from "./AuthContext"; import { Navigate } from "react-router-dom";

function Login() {

const [email, setEmail] = useState(""); const [password, setPassword] = useState("");

const { login, isLoggedIn } = useContext(AuthContext);

const handleSubmit = async (event) => { event.preventDefault();

// Fake network request for JWT token const jwtToken = "fake-jwt-token"; login(jwtToken);

};

if (isLoggedIn) {

return <Navigate to="/" />;

}

//...

Here we are checking if the user is logged in, and if they are, we redirect them to the home page using the Navigate component from react-router-dom. We can use a similar technique to redirect the user to the login page if when they are **not** logged in.

# Adding a Protected Route

Now that we have our AuthProvider and routes set up, let's add a protected route to our app. This route will only be accessible to authenticated users. We'll create a Profile component that will be our protected route.

## Creating the Profile Component

Create a new file src/Profile.jsx and add the following content:

import { useContext } from "react";

import { AuthContext } from "./AuthContext";

function Profile() {

const { token } = useContext(AuthContext);

return (

<div>

<h1>Profile</h1>

<p>Your secret token is: {token}</p>

</div>

);

}

export default Profile;

In the Profile component above, we:

1. Import the necessary hooks and AuthContext.
2. Use the useContext hook to access the token from our AuthContext.
3. Render the user's secret token as part of their profile information.

## Creating a Route Guard

To protect the Profile route, we'll create a higher-order component called RouteGuard that will check if the user is logged in before rendering the protected component.

Create a new file src/RouteGuard.jsx and add the following content:

import { useContext } from "react";

import { Navigate } from "react-router-dom"; import { AuthContext } from "./AuthContext";

function RouteGuard({ children }) {

const { isLoggedIn } = useContext(AuthContext);

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

}

export default RouteGuard;

In the RouteGuard component above, we:

1. Import the necessary hooks, Navigate from react-router-dom, and

AuthContext.

1. Use the useContext hook to access the isLoggedIn value from our

AuthContext.

1. Check if the user is logged in, and if so, render the protected component. Otherwise, redirect the user to the /login page.

## Adding the Protected Route

Finally, let's add the Profile component as a protected route in our App

component.

Modify src/App.jsx to include the Profile route and wrap it with the RouteGuard:

import { BrowserRouter, Link, Route, Routes } from "react-router-dom"; import { AuthProvider } from "./AuthContext";

import Home from "./Home"; import Login from "./Login"; import Profile from "./Profile";

import RouteGuard from "./RouteGuard";

function App() { return (

<AuthProvider>

<BrowserRouter>

<nav>

<ul>

<li>

<Link to="/">Home</Link>

</li>

<li>

<Link to="/login">Login</Link>

</li>

<li>

<Link to="/profile">Profile</Link>

</li>

</ul>

</nav>

<main>

<Routes>

<Route path="/" element={<Home />} />

<Route path="/login" element={<Login />} />

<Route

path="/profile" element={

<RouteGuard>

<Profile />

</RouteGuard>

}

/>

</Routes>

</main>

</BrowserRouter>

</AuthProvider>

);

}

export default App;

Now, the Profile route is protected by the RouteGuard component. Users who aren't logged in will be redirected to the /login page when trying to access the

/profile route.

With this setup, you've successfully implemented a protected route in your

React app using AuthProvider and react-router-dom. Not only can you share data between pages with ease, but you can also create a more secure user experience by protecting sensitive routes.

# Enhancing the User Experience with a Dynamic Navigation Bar

To make our app even more engaging and user-friendly, let's create a dynamic navigation bar that updates its content based on the user's logged-in status.

Modify the src/App.jsx file to include a separate Navigation component:

import { useContext } from "react";

import { BrowserRouter, Link, Route, Routes } from "react-router-dom"; import { AuthProvider, AuthContext } from "./AuthContext";

import Home from "./Home"; import Login from "./Login"; import Profile from "./Profile";

import RouteGuard from "./RouteGuard";

function Navigation() {

const { isLoggedIn } = useContext(AuthContext);

return (

<nav>

<ul>

<li>

<Link to="/">Home</Link>

</li>

{isLoggedIn ? (

<>

<li>

<Link to="/profile">Profile</Link>

</li>

</>

) : (

<li>

<Link to="/login">Login</Link>

</li>

)}

</ul>

</nav>

);

}

function App() { return (

<AuthProvider>

<BrowserRouter>

<Navigation />

<main>

<Routes>

<Route path="/" element={<Home />} />

<Route path="/login" element={<Login />} />

<Route

path="/profile" element={

<RouteGuard>

<Profile />

</RouteGuard>

}

/>

</Routes>

</main>

</BrowserRouter>

</AuthProvider>

);

}

export default App;

In the Navigation component above, we:

1. Import the necessary hooks and AuthContext.
2. Use the useContext hook to access the isLoggedIn value from our

AuthContext.

1. Conditionally render the "Profile" or "Login" link based on the user's logged-in status.

Now, the navigation bar will display the "Profile" link only when the user is logged in. Otherwise, it will show the "Login" link.

# Storing and Loading JWT Token from Local Storage

In this section, we'll enhance our app by storing the JWT token in the browser's local storage. This way, the user's logged-in status will persist across sessions, and they won't have to log in again every time they visit the site.

First, let's update our AuthContext to store the JWT token in local storage when the user logs in and load the token from local storage when the context is initialized.

Modify src/AuthContext.jsx to include local storage handling.

import { createContext, useContext, useState, useEffect } from "react"; const AuthContext = createContext();

function AuthProvider({ children }) {

const [isLoggedIn, setIsLoggedIn] = useState(false); const [jwt, setJwt] = useState(null);

const [isLoading, setIsLoading] = useState(true);

useEffect(() => {

const storedJwt = localStorage.getItem("jwt"); if (storedJwt) {

setIsLoggedIn(true); setJwt(storedJwt);

}

setIsLoading(false);

}, []);

function login(token) { setIsLoggedIn(true); setJwt(token); localStorage.setItem("jwt", token);

}

function logout() { setIsLoggedIn(false); setJwt(null); localStorage.removeItem("jwt");

}

const authValue = { isLoggedIn, isLoading, login,

logout,

};

return <AuthContext.Provider value={authValue}>{children}

</AuthContext.Provider>;

}

export { AuthProvider, AuthProvider };

In the updated AuthProvider component, we've made the following changes:

1. Added a useEffect hook that runs when the component is mounted. This hook checks if there's a JWT token stored in local storage, and if so, it sets the isLoggedIn state to true and updates the jwt state with the stored token.
2. Modified the login function to store the JWT token in local storage when the user logs in.
3. Modified the logout function to remove the JWT token from local storage when the user logs out.

When the user logs in, the JWT token will be stored in local storage, and their logged-in status will persist across browser sessions. When they visit the site again, their logged-in status will be automatically restored using the stored token.

Since useEffect runs after the component is mounted, we won't know if the user is logged in until components have already been loaded. So we need to include an isLoading state to indicate that the user's logged-in status is still being determined.

Modify src/RouteGuard.jsx to use isLoading.

import { useContext } from "react";

import { Navigate } from "react-router-dom"; import { AuthContext } from "./AuthContext";

function RouteGuard({ children }) {

const { isLoggedIn, isLoading } = useContext(AuthContext);

if (isLoading) { return <></>;

}

if (!isLoggedIn) {

return <Navigate to="/login" />;

}

return children;

}

export default RouteGuard;

1. If the localStorage hasn't been checked yet, isLoading will be true and we'll return an empty fragment to "do nothing".
2. If the user is not logged in, we'll redirect them to the login page.
3. Otherwise, the user is logged in and we'll render the child components.

# Final Code

<https://github.com/Sam-Meech-Ward/react-router-auth-context>

# Summary

In this tutorial, we've built a React 18 app using Vite that can handle user authentication using the useContext hook. We've created an AuthContext to manage the logged-in status and JWT token, a Login component to handle user logins, and a Home component to display user status. We also added a protected route and a secret page to demonstrate how to restrict access based on the user's logged-in status.

As you can see, the useContext hook is a powerful tool for managing and sharing state across your app. While this tutorial focused on user authentication, the concepts can be applied to many other use cases as well.

Find an issue with this page? [Fix it on GitHub](https://raw.githubusercontent.com/meech-ward/sammeechward.com_mdx/master/content/articles/react/use-context-auth/index.mdx)