🍿 7 min. read

Creating Protected Routes In NextJS With Supabase

Monica Powell

Follow @indigitalcolor 9,704 followers

🌻

> This articles walks through how to create protected routes on NextJS with Supabase's user management. It assumes you already have a NextJS site up and running with the ability to create new Supabase users but if not check out the first part of this series on **creating new Supabase users in NextJS**

Table of Contents

Supabase Auth Overview

Supabase has various methods in their JavaScript client library to handle user authentication and uses JSON Web Tokens (JWT) under the hood to manage authentication. If you want to learn more about how Auth works in Supabase check out the **Supabase auth deep-dive video series**. In order to have protected routes on our NextJS site, we'll need a way to register and authenticate users. We can perform these user actions and checks with the following methods from the **Supabase Auth client**. :

- **supabase.auth.signUp** - We should give users the ability to create an account (covered in the first article on **creating new Supabase users in NextJS**)
- **supabase.auth.signIn** - We need to give users the ability to sign-in. In this particular article, we'll cover the traditional method of using a username and password for sign-in but Supabase also supports other ways to log in, including OAuth providers (GitHub, Google, etc.) and magic links.

- **supabase.auth.user** - We need a way to determine if a user is currently logged-in in order to ensure that logged-out users are not able to view pages that should only be accessible to logged-in users and that the proper information is displayed in various places like the site navigation.
- **supabase.auth.signOut** - We should give users the ability to sign out and unauthenticate their session.

Create Protected Route

In order to create a protected route we need to have a particular page component we'd like to protect. For this example let's created a protected page at `pages/protected.js` that we can view at `localhost:3000/protected` when our site is running locally. This protected page will make a fetch request to a `getUser` API route to determine if there is currently an authenticated user loading the page. The API call should return the current user when there is one. We can then use this API response to redirect the page to the login page when there is no current user and only display user-specific information on the protected route when there is a user.

The API request can be made with `getServerSideProps()` which is a NextJS function that is called before a page renders. This allows us to redirect before the page renders based on the response from the `getUser` API call.

```
JS
1  import { basePath } from "../utils/siteConfig";
```

```
2
3  export async function getServerSideProps() {
4      // We need to implement `/api/getUser` by creating
5      // an endpoint in `pages/api` but for now let's just call it
6    const response = await fetch(`${basePath}/api/getUser`).then((response) =>
7      response.json()
8    );
9
10   const { user } = response;
11
12  // If the `getUser` endpoint doesn't have a user in its response
13  // then we will redirect to the login page
14  // which means this page will only be viewable when `getUser` returns a user.
15
16   if (!user) {
17     return {
18       redirect: { destination: "/login", permanent: false },
19     };
20   }
21   // We'll pass the returned `user` to the page's React Component as a prop
22   return { props: { user } };
23 }
24 export default function Protected({ user }) {
25   return (
26           <p>
27           // Let's greet the user by their e-mail address
```

```
            Welcome {user.email}!{" "}
            <span role="img" aria-label="waving hand">
              👋🏾
            </span>{" "}
          </p>{" "}
          You are currently viewing a top secret page!
  );
}
```

⚠

In this instance, NextJS requires absolute paths for the API routes and if you do not have an absolute route then you'll receive the following error:**"Error: only absolute urls are supported"**. In order to resolve this I created a helper function in **utils/siteConfig** to set the basePath based on the environment. In order for this to work there needs to be a **PRODUCTION_URL** set in your deployed site's environment variables.

JS

```
const dev = process.env.NODE_ENV !== "production";
export const basePath = dev ? "http://localhost:3000" : process.env.PRODUCTION_
```

Now, we need to actually implement the `getUser` API route that the protected route is calling by creating a file `pages/api/getUser.js`. Within this file we will make a request to `supabase.auth.user()` which returns the current user when there is a user currently logged-in.

```JS
import { supabase } from "../../utils/supabaseClient";

export default async function getUser(req, res) {
  const user = await supabase.auth.user();
  return res.status(200).json({ user: user });
}
```

The above code assumes that you've already set up a Supabase Client which we covered in the **first post of this series**. The Supabase client we are using in this instance looks like the below and uses environment variables to determine the Supabase DB URL and associated key:

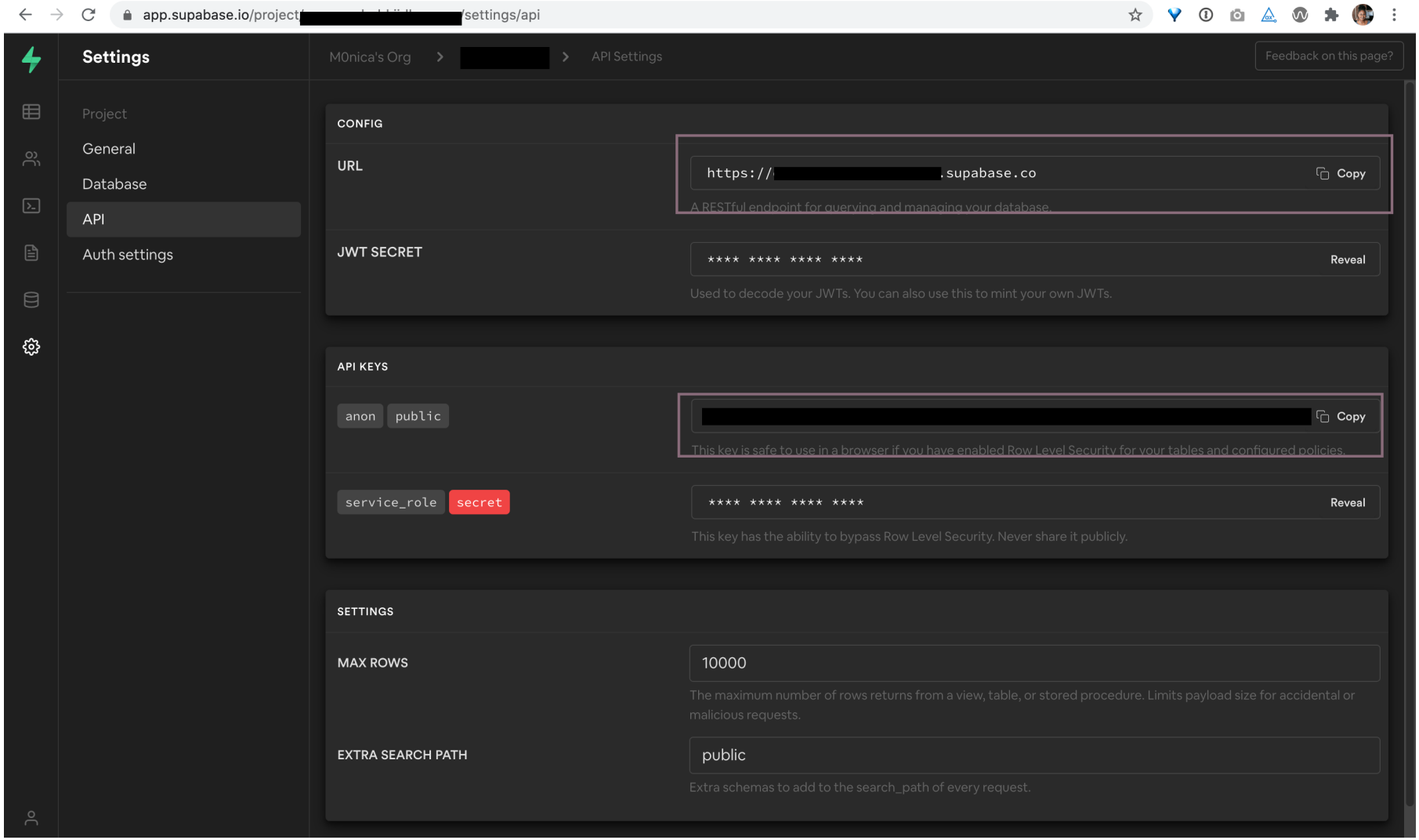
```JS
```

```
import { createClient } from "@supabase/supabase-js";

const supabaseUrl = process.env.SUPABASE_URL;
const supabaseKey = process.env.SUPABASE_KEY;

export const supabase = createClient(supabaseUrl, supabaseKey);
```

You can retrieve the API key and database URL associated with your Supabase project from `https://app.supabase.io/project/yourprojecturl]/settings/api` which can be navigated to by going to your project > settings > API.



a screenshot of the Supabase settings page

Sign-in and Redirect to Protected Page

We'll allow folks to log in and log out of the site using the sitewide navigation. In order to show the appropriate links based on authentication status, we can use the state to track if a user is currently authenticated. As a default, we'll set authentication status to `false` so that the navigation defaults to thelogged-out view.

When a user is authenticated then we will show the Sign Out text in the nav:

If there is no authenticated user then we will link to the Sign-in and Sign Up pages:


```JS
import Link from "next/link";
import { useEffect, useState } from "react";

export default function Header() {
  const router = useRouter();
  // Let's use state to track if a user is currently authenticated
  // As a default we'll set this value to false so that the navigation defaults to
```

```
  const [isAuthed, setAuthStatus] = useState(false);

// We'll set up the nav, on mount to call the getUser endpoint we just
// created to determine if a user is currently logged-in or not
  useEffect(() => {
    fetch("./api/getUser")
      .then((response) => response.json())
      .then((result) => {
        setAuthStatus(result.user && result.user.role === "authenticated");
      });
  }, []);

  return (

      <nav>
        <div>
        // If user is authenticated then we will show the Sign Out text
          {isAuthed ? (
            <span>
              <h3>Sign Out &rarr;</h3>
            </span>
          ) : (
              // If there is no authenticated user then we will link to the Sign-in
            <>
              <Link href="/signup">
                <h3>Sign Up &rarr;</h3>
```

```
              </Link>
              <Link href="/login">
                <h3>Login &rarr;</h3>
              </Link>
            </>
          )}
        </div>
      </nav>
  );
}
```

When a user clicks "Sign In" from the nav we will navigate the user to the `login` page which contains a form to allow users to sign-in. The form will collect a user's email and password and on submit will fire a function `signInUser` which makes an API request to an API route for `login` and passes the `email` and `password` values from the form submit event to the API. If all goes well then we will receive a user object and can redirect (using NextJS's client-side router) to the `/protected` route that serves as a landing page for logged-in users.

JS

```
import { useRouter } from "next/router";

```

```
export default function Form() {
  const router = useRouter();
  const signInUser = async (event) => {
    event.preventDefault();

    const res = await fetch(`/api/login`, {
      body: JSON.stringify({
        email: event.target.email.value,
        password: event.target.password.value,
      }),
      headers: {
        "Content-Type": "application/json",
      },
      method: "POST",
    });

    const { user } = await res.json();
    if (user) router.push(`/protected`);
  };

  return (
    <form onSubmit={signInUser}>
      <label htmlFor="email">Email</label>
      <input
        id="email"
        name="email"
```

```
        type="email"
        autoComplete="email"
        required
      />
      <label htmlFor="password">Password</label>

      <input
        type="password"
        id="password"
        name="password"
        required
      />
      <button type="submit">Login</button>
    </form>
  );
}
```

The `login` API route will use `supabase.auth.signIn` to sign-in a user. If a user is successfully signed in then the API will return a 200 response, or else the API will return a 401 response. The form is not yet set up to handle this 401 response but ideally, we'd want to return some type of message to the user informing them that their credentials were invalid and prompt them to attempt to sign-in again or reset their password. However, as this

app is currently being built the functionality to reset password does not yet exist so this error path cannot be fully handled yet.

```JS
import { supabase } from "../../utils/supabaseClient";

export default async function registerUser(req, res) {
  const { email, password } = req.body;
  let { user, error } = await supabase.auth.signIn({
    email: email,
    password: password,
  });
  if (error) return res.status(401).json({ error: error.message });
  return res.status(200).json({ user: user });
}
```

Sign Out and Redirect to Homepage

Let's update the Sign Out link in the header to be functional by creating a `signOut` function that fires on click of the Sign Out text.

```JS
<span onClick={signOutUser}>
   <h3>Sign Out &rarr;</h3>
</span>
```

We'll also want to import a router from `next/router` to handle our client-side redirect.

```JS
import { useRouter } from "next/router";
```

For `signOutUser` let's make a call to a `logout` API route that sets the `authStatus` to `false` when a user is successfully signed out. We also want to ensure that when a user is not logged-in they are not viewing an authenticated page by redirecting to the homepage if a user logs out on a page other than the homepage. Without explicitly redirecting to the homepage when a user signs out, the state of `authStatus` would change in

the nav as well as the logged-in vs.logged-out specific text however, the actual page regardless of authentication would continue showing protected information for unauthenticated users which we don't want.

```JS
const signOutUser = async () => {
    const res = await fetch(`/api/logout`);
    if (res.status === 200) setAuthStatus(false);
    // redirect to homepage when logging out users
    if (window.location !== "/") router.push("/");
  };
```

Now we need to create the `/api/logout` route so that we can actually use it when the `signOutUser` function fires.

```JS
import { supabase } from "../../utils/supabaseClient";

export default async function logoutUser(req, res) {
```

```
  let { error } = await supabase.auth.signOut();

  if (error) return res.status(401).json({ error: error.message });
  return res.status(200).json({ body: "User has been logged out" });
}
```

Summary

So in conclusion, we created a protected route by creating a page component in NextJS that calls a `getUser` endpoint in `getServerSideProps()` and redirects to the log in page, instead of loading the protected route, when there is not a user returned. We also set up client-side routing to redirect users to `/protected` when they successfully logged-in and to the homepage `/` when they logged out. The core functionality to update and check authentication was handle in API routes using Supabase's various auth methods (signIn, signOut, user).

Example Code on GitHub

You can view the full source code for the example code at: **https://github.com/M0nica/protected-routes-with-supabase-nextjs-example**

Looking Ahead

I am looking forward to sharing more about the app development as I progress through my journey of developing **Shine Docs** . As I wrap up the authentication for this site I am considering adding additional functionality like magic links or other auth providers, which are natively supported by Supabase. Before I extend the auth functionality to support additional ways of authenticating I will need to update the site to give users the ability to reset their own password and better handle authentication errors to ensure that the sign-in (are the user credentials invalid? did something go wrong during sign-in?) and sign up (has an e-mail already been claimed? is a password not secure enough?) flow are as seamless as possible.

This article was published on March 13, 2021.

NextJS **Supabase**

Subscribe

I won't send you spam. Unsubscribe at *any* time.

Webmentions 10 3 Show Mentions

That's All!

Thanks for reading "**Creating Protected Routes In NextJS With Supabase**"

Older Post

Creating New Supabase Users In NextJS →

"You can't use up creativity.
The more you use, the more you have."

— Dr. Maya Angelou

Interested in collaborating? **Send me a message** • Want to support my work? **Become a GitHub Sponsor**

Made with ♥, </>, & **various tech** in New York City