5 - Authentication

In the previous chapters, we've created a simple web application that pulls data from a database and displays it with various types of visualizations.
In this chapter, we will ensure that only authorized users can access our application.

Though we won't be addressing how to go live with this web application, we will still implement a basic authentication system using GitHub OAuth, just to see how we could secure our application should we decide to deploy it in the future.

Chapter overview

Learning goals

By the end of this chapter, you will be able to:

- Understand the basics of authentication and authorization
- Implement a simple authentication system using GitHub OAuth
- Create a login page in Svelte
- Restrict access to certain routes based on authentication status in SvelteKit
- Be able to whitelist users to access the application

Learning resources

Auth.js

We'll be using Auth.js to implement authentication in our SvelteKit application. Auth.js is a flexible authentication library that supports various OAuth providers, including GitHub. It allows us to easily integrate authentication into our application without having to build everything from scratch.

Why a library over manual implementation?

Security is a critical aspect of web applications, yet setting up proper, bulletproof authentication can get quite complex.
It is therefore outside of the scope of this course to cover all the intricacies of authentication, and consequently, we will be looking more at how we can implement authentication in our SvelteKit application, rather than the details of how authentication works in general.
We'll go over some basic concepts of authentication and authorization, but we won't dive deep into the security aspects of OAuth or how to implement it yourself securely.

If you do want to learn more about authentication, then Lucia is a great resource to learn about how to manually roll out authentication, providing a more in-depth understanding of the topic.

What is OAuth and why use it?

If you've ever seen a "Log in with Google" or "Log in with GitHub" button on a website, you've encountered OAuth. It allows users to grant access to their resources on an external service, like Google, to your application without sharing their credentials (from: The Copenhagen Book). By offhanding the authentication process to a trusted provider, we don't have to worry about securely storing passwords in our database.

Auth.js recommends using OAuth as the primary method of authentication, as it saves us from a lot of complexity that comes with managing user credentials and security:

> OAuth services spend significant amounts of money, time, and engineering effort to build abuse detection (bot-protection, rate-limiting), password management (password reset, credential stuffing, rotation), data security (encryption/salting, strength validation), and much more. It is likely that your application would benefit from leveraging these battle-tested solutions rather than try to rebuild them from scratch. Or if you don’t want to pay for an OAuth service, we support many self-hosted OAuth providers like Keycloak.

How our application will authenticate users

In our `.env` file, we will store a whitelist of GitHub users that are allowed to log in to our application.
We will use GitHub OAuth to authenticate users. When a user logs in, we will check if their GitHub userID is in this whitelist.
If they are whitelisted, they will be granted access to the application by storing their 'session' in a cookie (more on this later).
We can then use this session cookie in our `+page.server.ts` files to check if a user should be allowed to access certain routes or not.
If they are not whitelisted, they will not be able to authenticate, and won't get this 'session cookie'. In our server, we can then redirect those users to a "not authorized" page, rather than serving the protected content.

An 'explain like I'm five version' of the authentication process goes as follows.

Say our application is a super secret clubhouse with awesome toys, and only certain kids are allowed in.

Clubhouse Analogy	**Technical Implementation**
Johnny wants to play in the clubhouse and knocks on the door	User visits a protected route/URL in your application
"Sorry, you can't come in without a wristband, please go get one across the street at Mr. GitHub's house"	Auth.js middleware detects unauthenticated user and redirects to GitHub OAuth authorization endpoint
Johnny goes to GitHub house and clicks "Log in with GitHub"	User is redirected to `https://github.com/login/oauth/authorize` with your app's client ID
Mr. GitHub asks Johnny to prove they're really Johnny, Johnny shows ID (password)	GitHub prompts user to enter their GitHub username/password or use existing session
Mr. GitHub gives Johnny a sealed envelope with a special number inside	GitHub redirects back to your app with an authorization code in the callback URL
Johnny takes envelope back to clubhouse door	User's browser follows the redirect back to your app's callback route
We open the envelope and see "Yep, this is Johnny! His number is 12345"	Your app exchanges the authorization code for an access token, then uses it to fetch user's GitHub profile (including user ID)
We check our list of kids allowed in the clubhouse	Your app checks if the GitHub user ID exists in your `.env` whitelist
We give Johnny a red wristband with their name on it	Auth.js creates a session cookie with user information
Johnny shows wristband to use toys	User's subsequent requests include the session cookie for authentication
We take back the wristband when Johnny leaves	User logs out, session cookie is cleared/invalidated
Wristband expires after some time (tomorrow only blue wristbands work)	Session cookie has expiration time, user must re-authenticate
We can remove Johnny's number from our list if they misbehave	Admin can remove user's GitHub ID from the whitelist in `.env`

In this analogy, we can also think of the clubhouse as a place where they may access some of our toys (public routes), but only the kids with a wristband can play with the rest of our toys (protected routes).

With some basic understanding of how our authentication will work, let's get started with implementing it in our SvelteKit application!

Setting up Auth.js

We'll be following along with the Auth.js documentation to set up authentication in our SvelteKit application. Try to follow the latest instructions from Auth.js itself, as it may change over time, but we will describe the steps we took to set it up in this chapter.

First, we install the `auth.js` dependency:

```
npm i @auth/sveltekit
```

Next, we create an `AUTH_SECRET` variable in our `.env` file. This is a randomly generated string that only we will know, and thus we can use it for encrypting our session cookies. Auth.js can generate this secret for us:

```
npx auth secret
```

(Don't forget to update your `.env.example` file with a new blank `AUTH_SECRET` variable (DO NOT store the actual secret in your `.env.example` file, as this will be pushed to our repository!))

Next, we'll create a config file for Auth.js.

```
// /src/lib/server/auth/auth.ts

import { SvelteKitAuth } from "@auth/sveltekit"

export const { handle } = SvelteKitAuth({
  providers: [],
})
```

We want to re-export this `handle` function in `src/hooks.server.ts`, which ensures that this `handle` we just created is used for all requests in our SvelteKit application (see Server Hooks).

```
// /src/hooks.server.ts

```

```
export { handle } from "./auth"
```

This `handle` function adds a method `auth()` to [the `event.locals` object](), which we can access in any `+page.server.ts` or `+layout.server.ts` file, by adding the `locals` parameter to the `load` function.

For example, in `src/routes/+layout.server.ts`, we can access the `auth` method like this:

```
// /src/routes/+layout.server.ts

import type { LayoutServerLoad } from './$types';

export const load: LayoutServerLoad = async (event) => {
    const session = await event.locals.auth();

    return {
        session
    };
};
```

Now, we need to add the GitHub OAuth provider.

First, let's [create a GitHub OAuth application]().

- Go to your GitHub account settings
- Navigate to "Developer settings" (left sidebar, at the very bottom)
- Click on "OAuth Apps" (left sidebar)
- Click on "New OAuth App"
- Give your application a name, e.g. "oauth-demobots"
- Set the "Homepage URL" to `http://localhost:5173` (this is where the app runs locally)
- Set the "Authorization callback URL" to `[origin]/auth/callback/github` (so for us, it will be `http://localhost:5173/auth/callback/github`)
- Click "Register application"

Next, take note of the "Client ID" and generate a new "Client secret", and add these to your `.env` file. While we're at it, let's also add a variable for the `ALLOWED_GITHUB_IDS`, which will be a comma-separated list of GitHub user IDs that are allowed to log in to our application.

Your `.env.example` file should look like this, and your `.env` file should be similar, but with your actual credentials filled in:

```
.env.example

# Replace with your DB credentials!
DATABASE_URL="mysql://username:password@host:port/dbname"
# Auth.js configuration
AUTH_SECRET=""
AUTH_GITHUB_ID=""
AUTH_GITHUB_SECRET=""
# Comma-separated list of allowed GitHub user IDs (not usernames)
# Get yours at, e.g., https://api.github.com/user/codevogel
ALLOWED_GITHUB_IDS=""
```

Now, let's add the GitHub provider to our Auth.js configuration:

```
// /src/lib/server/auth/auth.ts

import { SvelteKitAuth } from "@auth/sveltekit"
import GitHub from "@auth/sveltekit/providers/github"

export const { handle, signIn } = SvelteKitAuth({
  providers: [GitHub],
})
```

We're pretty much done with the setup of Auth.js, but we still need to implement some sign in/out logic, and logic that protects our routes from users who aren't logged in.

Adding a Sign In route

Now, we will add a server-side login route that handles the `POST` requests to sign in users:

We will add the `signIn` function to our exports in our Auth.js configuration file:

```
// /src/lib/server/auth/auth.ts
...

export const { handle, signIn } = ...
})
```

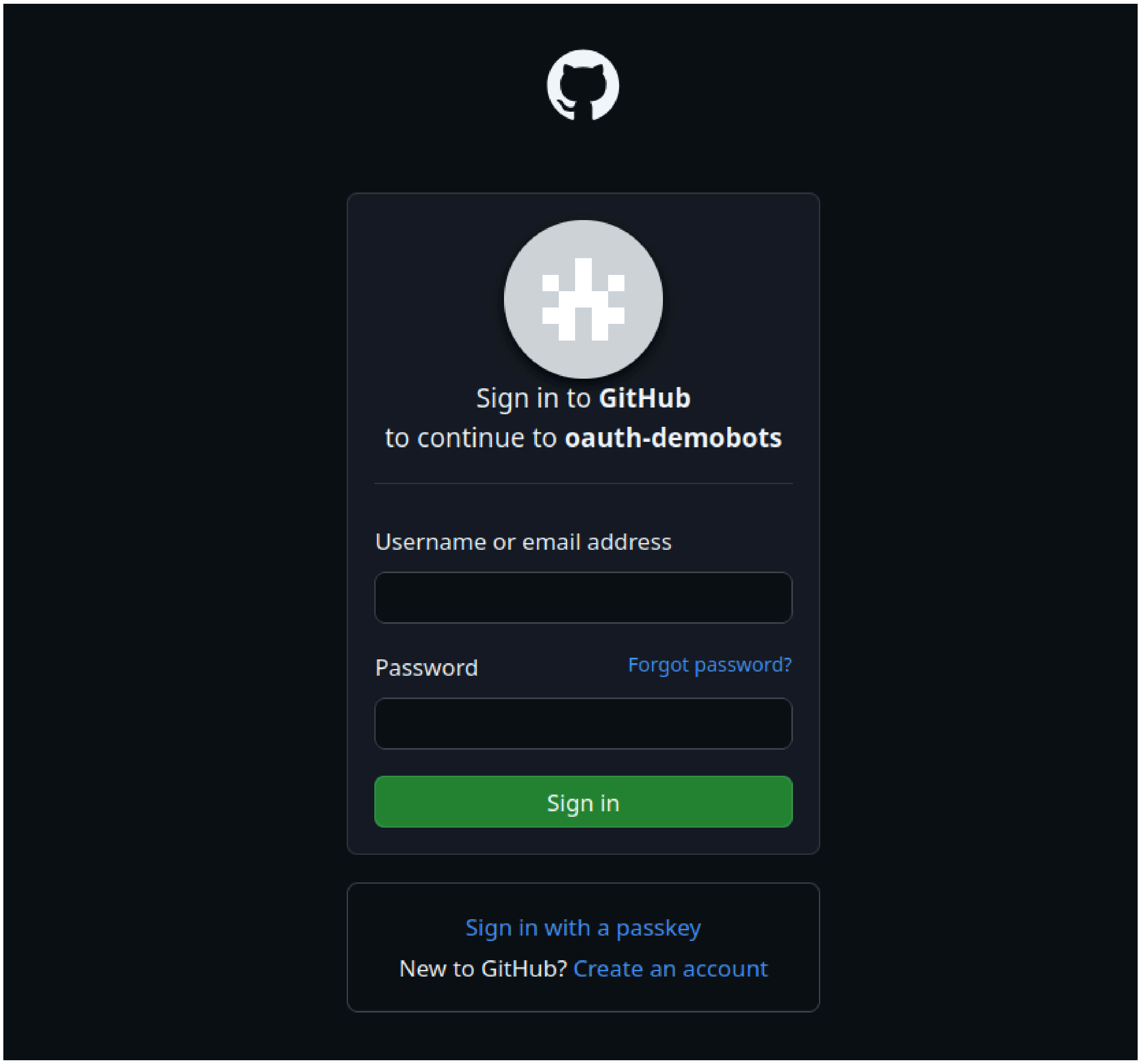
Then we will create the `+server.ts` file in the `src/routes/auth/sign-in` directory, which will handle the sign-in requests. This file will export an `actions` object with a `default` action that calls the `signIn` function we just exported.

```
// /src/routes/auth/sign-in/+server.ts

import { signIn } from "$lib/server/auth/auth"
import type { Actions } from "./$types"
export const actions: Actions = { default: signIn }
```

And next, we'll add a Sign In button to our application.
For now, we'll just add it to a new page at `/src/routes/login/+page.svelte`. Don't worry about the styling for now, we'll move the Sign In button to a more appropriate place later on.

```
<!-- src/routes/login/+page.svelte -->
<script lang="ts">
    import { SignIn } from '@auth/sveltekit/components';
    import { page } from '$app/state';
</script>

<div>
    <SignIn
        provider="github"
        signInPage="auth/sign-in"
    />
</div>
```

You can try to sign in to your application now, by navigating to `http://localhost:5173/login` in your browser.
This should redirect you to GitHub, where you are asked to sign in and/or authorize our OAuth application, and then redirect you back to our web app. This stores your session in a cookie, which is then used to authenticate you in subsequent requests.


Sign in to GitHub
to continue to oauth-demobots
Username or email address
Password
Forgot password?
Sign in
Sign in with a passkey
New to GitHub? Create an account

Of course, this doesn't do much on it's own just yet, as we haven't really implemented any logic to check if the user is allowed to access our application.
So, let's do that next.

Checking if a user is allowed to access the application

We want to secure all our routes under `/dashboard`. Let's start with the `/dashboard` route itself, and then we'll secure the other routes as well.

In `src/routes/dashboard/+page.server.ts`, we need to check if the user is authenticated. As said before, we can use the `auth()` method from `event.locals` to get the session information of the user.

We add the `locals` parameter to the `load` function, and then check if the `session` object has a `user` property. If it does, we load the dashboard data as normal. If it doesn't, we return a 401 error (401 indicates "Unauthorized") with a message indicating that the user is not logged in.

```
// /src/routes/dashboard/+page.server.ts

...
import { error } from '@sveltejs/kit';

export const load: PageServerLoad = async ({ locals }) => {
    const session = await locals.auth();

    if (!session?.user) {
        return error(401, 'You are not logged in. Please log in to access
this page.');
    }

    ...
};
```

Now, if you try to access `http://localhost:5173/dashboard`, you should still be able to see the dashboard data, as you are logged in (given you followed the sign-in steps above). If you haven't logged in yet, or visit the same page in an incognito window, you should instead see the "You are not logged in" error message.

Cool: Our dashboard route is now protected from users who aren't logged in!

Later, we'll also implement this in our other dashboard routes, but for now, let's add a sign out button to our application, so that users can log out of their session.

Adding a Sign Out route

Similar to the Sign In route, let's add a Sign Out route to our application.

First, expand the `src/lib/server/auth/auth.ts` file to export the `signOut` function:

```
// /src/lib/server/auth/auth.ts
...

export const { handle, signIn, signOut } = ...
```

Next, add the `+server.ts` file that handles the sign out requests:

```
// /src/routes/auth/sign-out/+server.ts
import { signOut } from "$lib/server/auth/auth"
import type { Actions } from "./$types"
export const actions: Actions = { default: signOut }
```

And then a `+page.svelte` file that displays the Sign Out button. Again, we will just add it to a new page at `/src/routes/logout/+page.svelte` for now, but we will move it to a more appropriate place later on, so don't worry about the styling for now.

```
<!-- /src/routes/logout/+page.svelte -->
<script lang="ts">
   import { SignOut } from "@auth/sveltekit/components"
</script>

<div>
    <SignOut
        provider="github"
        signOutPage="auth/sign-out"
    />
</div>
```

Try navigating to `http://localhost:5173/logout` in your browser, and you should see a Sign Out button. Click it, and you should no longer be able to access the `/dashboard` route, as you are now logged out, and your cookie session has been cleared (you can check this in your browser's developer tools, under the "Application" tab, in the "Cookies" section).

Okay, cool. We can sign in and out of our application, but any GitHub user can sign in. Let's start making some use of that `ALLOWED_GITHUB_IDS` variable in our `.env` file, and restrict access to our application to only those users.

Restricting access to whitelisted users

To restrict access to our application, we will check if the authenticated user's GitHub ID is in the `ALLOWED_GITHUB_IDS` list from our `.env` file.

In our Auth.js configuration file, `src/lib/server/auth/auth.ts`, we can add a callback function `signIn` to the `SvelteKitAuth` configuration method.
This function will be called after a user is redirected back to our application after signing in with GitHub, and we can use it to control whether that GitHub user is allowed to sign in to our web app.

In this function, we will check if the user's GitHub ID is in the `ALLOWED_GITHUB_IDS` list, and return `true` if it is, or `false` if it isn't. If the user is not allowed to sign in, we will redirect them to a "not whitelisted" page, using the `pages` option in the `SvelteKitAuth` configuration.

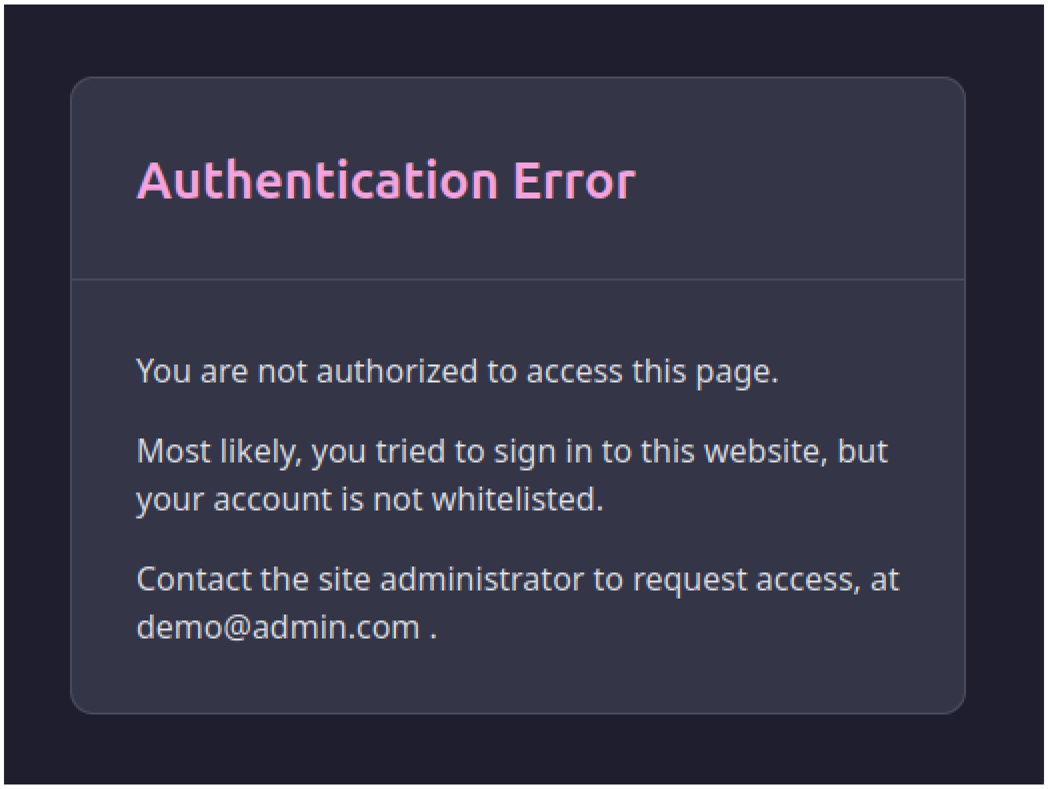
```
// /src/lib/server/auth/auth.ts

import { SvelteKitAuth } from '@auth/sveltekit';
import GitHub from '@auth/sveltekit/providers/github';
import { ALLOWED_GITHUB_IDS } from '$env/static/private';

export const { handle, signIn, signOut } = SvelteKitAuth({
    providers: [GitHub],
    callbacks: {
        async signIn({ account, profile }) {
            if (!account || !account.provider || !profile || !profile.id) {
                return false;
            }
            if (account.provider === 'github') {
                // Check if user's GitHub ID is in your whitelist
                const allowedIds = ALLOWED_GITHUB_IDS.split(',').map((id) =>
id.trim());
                return allowedIds.includes(profile.id.toString());
            }
            return false;
        }
    },
    pages: {
        error: '/auth/not-whitelisted'
    }
});
```

Now, we just need to create the `/auth/not-whitelisted` page that users will be redirected to if they are not allowed to sign in:

```
<script lang="ts">
    import Card from '$lib/ui/views/Card.svelte';
    import { ADMIN_EMAIL } from '$lib/constants/strings';
</script>

<div class="flex h-full items-center justify-center">
    <Card>
        {#snippet header()}
            <h1 class="text-2xl font-bold">Authentication Error</h1>
        {/snippet}
        {#snippet article()}
            <p>You are not authorized to access this page.</p>
            <p>
```

```
                Most likely, you tried to sign in to this website, but your
account is not whitelisted.
            </p>
            <p>Contact the site administrator to request access, at
{ADMIN_EMAIL} .</p>
        {/snippet}
    </Card>
</div>
```

If you sign out and remove your GitHub user ID from the `ALLOWED_GITHUB_IDS` list in your `.env` file, you should now be redirected to this page when trying to sign in:



Authentication Error

You are not authorized to access this page.

Most likely, you tried to sign in to this website, but your account is not whitelisted.

Contact the site administrator to request access, at demo@admin.com .

Awesome, we now have control over who can access our application: We can sign in as a user, and sign out again, and we can restrict access to only those users that are whitelisted in our `.env` file.

Securing the other routes

We want to add the same authentication check to all our dashboard routes, so that no route is left unprotected.

Let's add the same auth check to the following `+page.server.ts` files:

- `src/routes/dashboard/session/+page.server.ts`
- `src/routes/dashboard/session/[id]/+page.server.ts`

- `src/routes/dashboard/user/+page.server.ts`
- `src/routes/dashboard/user/[id]/+page.server.ts`

We won't go over the code for each of these files, as it is very similar to the code we added to `src/routes/dashboard/+page.server.ts`. Just add the `locals` parameter to the `load` function, call `await locals.auth()`, and check if the `session.user` property exists. If it doesn't, return a 401 error with a message indicating that the user is not logged in.

As we'll be duplicating the error message in the `error()` function, we extract it to a constant in `src/lib/constants/strings.ts`:

```
// src/lib/constants/strings.ts

export const NOT_LOGGED_IN_ERROR = 'You are not logged in. Please log in to
access this page.';
```

e.g. for `src/routes/dashboard/session/[id]/+page.server.ts`:

```
...
import { error } from '@sveltejs/kit';
import { NOT_LOGGED_IN_ERROR } from '$lib/constants/strings';

export const load: PageServerLoad = async ({ params, locals }) => {
    const authSession = await locals.auth();

    if (!authSession?.user) {
        return error(401, NOT_LOGGED_IN_ERROR);
    }

    ...
}
```

Awesome, our dashboard routes are now all protected, and users will be redirected to the "not logged in" error page if they try to access them without being authenticated.

A better authentication flow

While our authentication flow works, we can't really expect our users to manually navigate to the `/login` and `/logout` pages to sign in and out of our application.

Instead, we will create a `LogInButton` and `LogOutButton` component that will handle the sign in and sign out actions for us. We'll place them neatly in the `NavBar` component, so that users can easily sign in and out of our application.

Furthermore, we'll ensure that users are redirected to `/` after signing out.

Creating the LogInButton component

We'll wrap the `SignIn` component from Auth.js in a new `LogInButton` component, which will handle the sign in action for us.
We can style the button to match our application's design, and we'll also add an icon to it.

For the Sign In button, we'll also add an optional `primary` prop, which will determine whether the button should be styled as a primary button or as an icon button, this way, we can re-use it in the error pages which tell users to log in.

We show the text "Log In" on larger screens, but hide it on smaller screens, so that the button fits in the NavBar on smaller screens.

```
<script lang="ts">
    import { LogIn } from 'lucide-svelte';
    import { SignIn } from '@auth/sveltekit/components';

    let { primary = false } = $props();
</script>

<SignIn
    provider="github"
    class={primary ? 'btn preset-filled-primary-500' : 'btn-icon
hover:preset-tonal'}
    signInPage="auth/sign-in"
>
    <span slot="submitButton" class="flex gap-x-1">
        <span class="{primary ? '' : 'hidden'} text-sm lg:block">Log
In</span>
        <LogIn />
    </span>
</SignIn>
```

Creating the LogOutButton component

This is very similar to the `LogInButton` component, but we'll use the `SignOut` component from Auth.js instead. We don't need the `primary` prop here, as placing it in the NavBar is sufficient.

Notable is that we add the `options` prop to the `SignOut` component, with a `redirectTo` option set to `/`. This ensures that after signing out, the user is redirected to the home page of our application.

```
<script lang="ts">
    import { LogOut } from 'lucide-svelte';
    import { SignOut } from '@auth/sveltekit/components';
</script>

<SignOut
    provider="github"
    class="btn-icon hover:preset-tonal"
    signOutPage="auth/sign-out"
    options={{ redirectTo: '/' }}
>
    <span slot="submitButton" class="flex gap-x-1">
        <span class="hidden text-sm lg:block">Log Out</span>
        <LogOut />
    </span>
</SignOut>
```

Adding the buttons to the NavBar

First, we'll load the `loggedIn` state in the `/src/routes/+layout.server.ts` file:

```
// /src/routes/+layout.server.ts

import type { LayoutServerLoad } from './$types';

export const load: LayoutServerLoad = async (event) => {
    const session = await event.locals.auth();

    const loggedIn = session?.user ? true : false;

    return {
        session,
        loggedIn
    };
};
```

Next, we'll pass this prop to the `NavBar` component:

```
<!-- src/routes/+layout.svelte -->

<script lang="ts">
    ...

    let { children, data } = $props();
    const loggedIn = $derived(data.loggedIn);
</script>

<div class="...">
    <header>
        <NavBar {loggedIn} />
    </header>
    ...
</div>
```

Then, we'll expand the `NavBar` component to accept the `loggedIn` prop, and conditionally render the `LogInButton` or `LogOutButton` based on the user's authentication status.

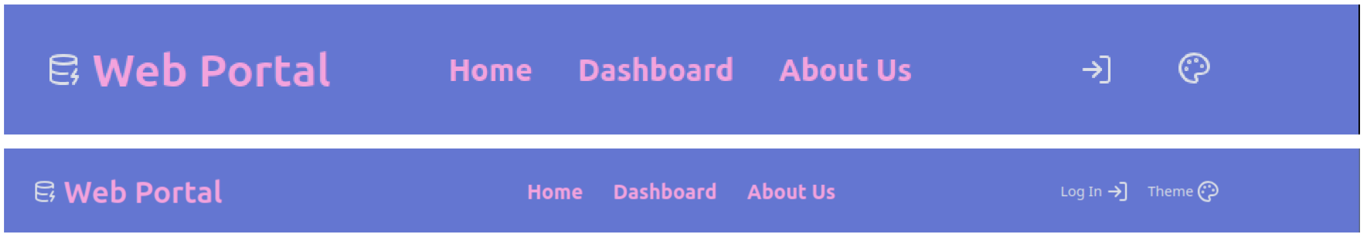
```
<!-- /src/lib/ui/nav/NavBar.svelte -->

<script lang="ts">
    ...
    import LogInButton from '$lib/ui/control/LogInButton.svelte';
    import LogOutButton from '$lib/ui/control/LogOutButton.svelte';

    let { loggedIn }: { loggedIn: boolean } = $props();

    ...
</script>

<div class="...">
    <div>...</div>
    <div>...</div>
    <div class="col-span-1 flex items-center md:gap-x-6">
        {#if loggedIn}
            <LogOutButton />
        {:else}
            <LogInButton />
        {/if}
        <ThemeSwitch />
    </div>
</div>
```

Just to stay consistent with these new buttons, we will also update the `ThemeSwitch` component to add some text next to the icon on larger screens:

```
<!-- /src/lib/ui/control/ThemeSwitch.svelte -->

```

```
<Popover ...>
    {#snippet trigger()}
        <span class="flex gap-x-1">
            <span class="hidden text-sm lg:block">Theme</span>
            <Palette />
        </span>
    {/snippet}
    ...
</Popover>
```

Now, you should see the Log In button in the NavBar when you are not logged in, and the Log Out button when you are logged in. Clicking the Log In button will redirect you to the GitHub OAuth flow, and clicking the Log Out button will log you out and redirect you to the home page.

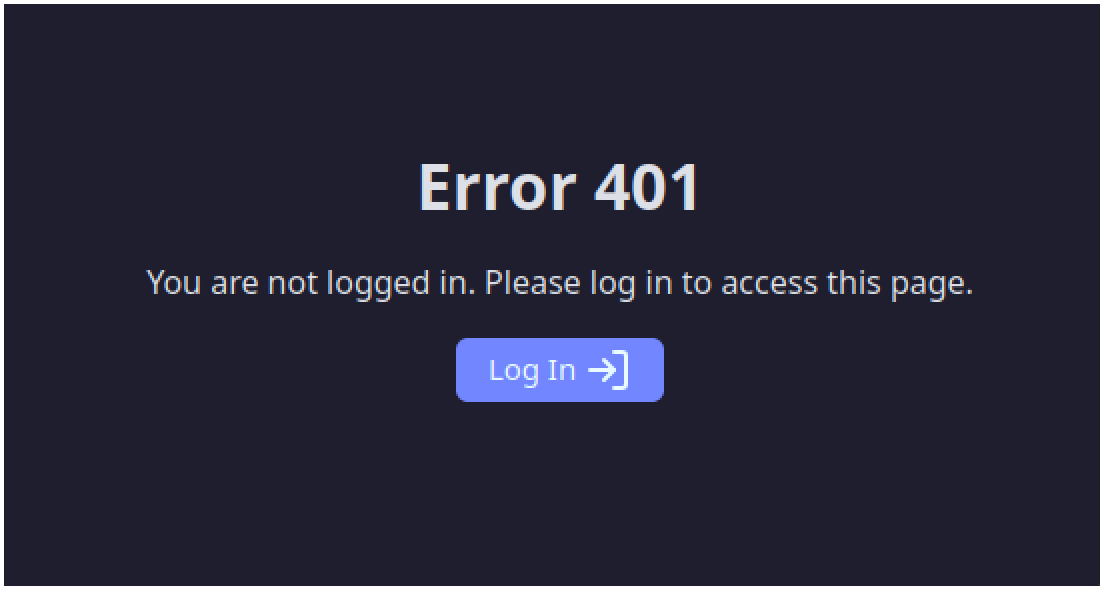


We can now remove the `/login` and `/logout` routes/pages, as we no longer need them (keep the `auth/sign-in` and `auth/sign-out` routes!). The Log In and Log Out buttons in the NavBar will handle the sign in and sign out actions for us.

Signing in from the error pages

Let's update our `Error.svelte` component to show a Log In button when the user encounters a 401 error, indicating that they are not logged in:

```
<!-- /src/lib/ui/views/Error.svelte -->

<script lang="ts">
    import LogInButton from '$lib/ui/control/LogInButton.svelte';
    let { status, message }: { status: number; message: string } = $props();
</script>

<div class="flex flex-grow flex-col items-center justify-center gap-y-4 p-
8">
    <h1 class="text-3xl font-bold">Error {status}</h1>
    <p class="text-center">{message}</p>
    {#if status === 401}
        <LogInButton primary={true} />
    {/if}
</div>
```


Error 401

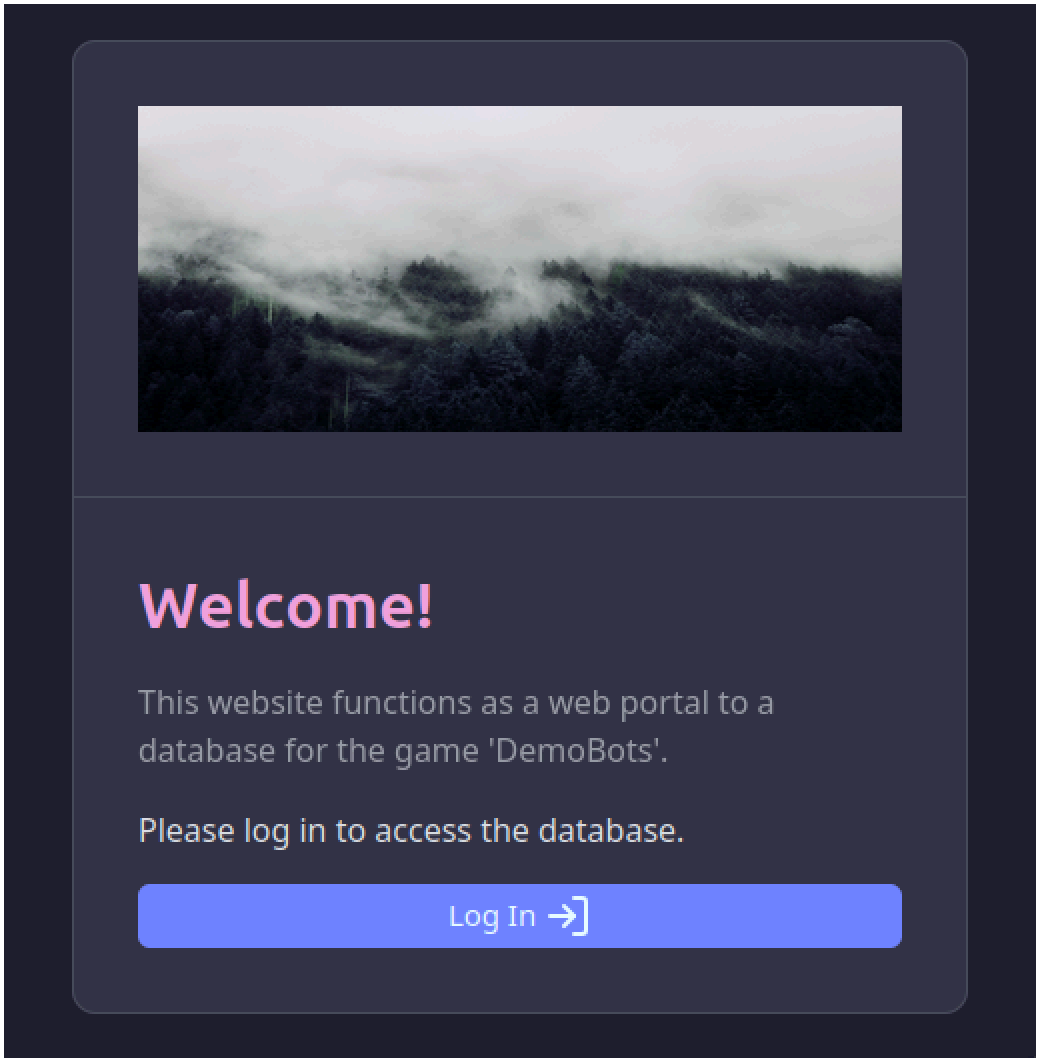
You are not logged in. Please log in to access this page.

Showing a Log In button on the home page when not logged in

Finally, let's add a Log In button to the home page, so that users can easily log in to our application from there as well.
We already added the `loggedIn` prop to our `+layout.server.ts` file, so we can use it in the home page as well:

```
<!-- /src/routes/+page.svelte -->

<script lang="ts">
    ...
    import LogInButton from '$lib/ui/control/LogInButton.svelte';

    let { data } = $props();
    const loggedIn = $derived(data.loggedIn);
</script>

<div
    class="..."
>
    <Card footerBase="...">
        ...
        {#snippet article()}
            ...
            {#if !loggedIn}
                <p>Please log in to access the database.</p>
                <LogInButton primary={true} />
            {/if}
        {/snippet}
    </Card>
</div>
```

The button should show up when not logged in:



Welcome!

This website functions as a web portal to a database for the game 'DemoBots'.

Please log in to access the database.

Wrapping up

Congratulations! You have successfully implemented a basic authentication system using GitHub OAuth and Auth.js in your SvelteKit application. You can now restrict access to your application to only whitelisted users, and provide a seamless login and logout experience.

You have learned:

- Understand the basics of authentication and authorization
- How to implement a simple authentication system using GitHub OAuth
- How to create a login page in Svelte
- How to restrict access to certain routes based on authentication status in SvelteKit
- How to whitelist users to access the application

This concludes the course on building a web application with SvelteKit. We hope you enjoyed it and learned a lot about SvelteKit, Svelte, and building web applications in general.