CMPS 350 Web Development Fundamentals - Spring 2023
Lab 12 – Securing Web applications: Authentication, Authorization, and Confidentiality

Objective

The objective of this lab is to practice how to secure your Next.js web application. You will practice how to:

- Add login functionality in your web app.
- Save passwords securely by hashing them using **Bycrpt**.
- Protect APIs using **JSON Web Token (JWT)**: an open standard (RFC 7519) that represents the user's identity and role as a compact and signed string that can be easily transmitted between the client and server.
- Use next-auth library for delegated authentication using identify providers such as Google and GitHub.
- Protect web pages and API routes from unauthorized access using next-auth middleware.
- Use sessions to store user identification and authentication credentials using different strategies such as JWT.

Project Setup

Download **Lab13-Web Security** from the GitHub Repo and copy it to your repository.
Open the **BankingApp** in VS Code and complete the tasks below.

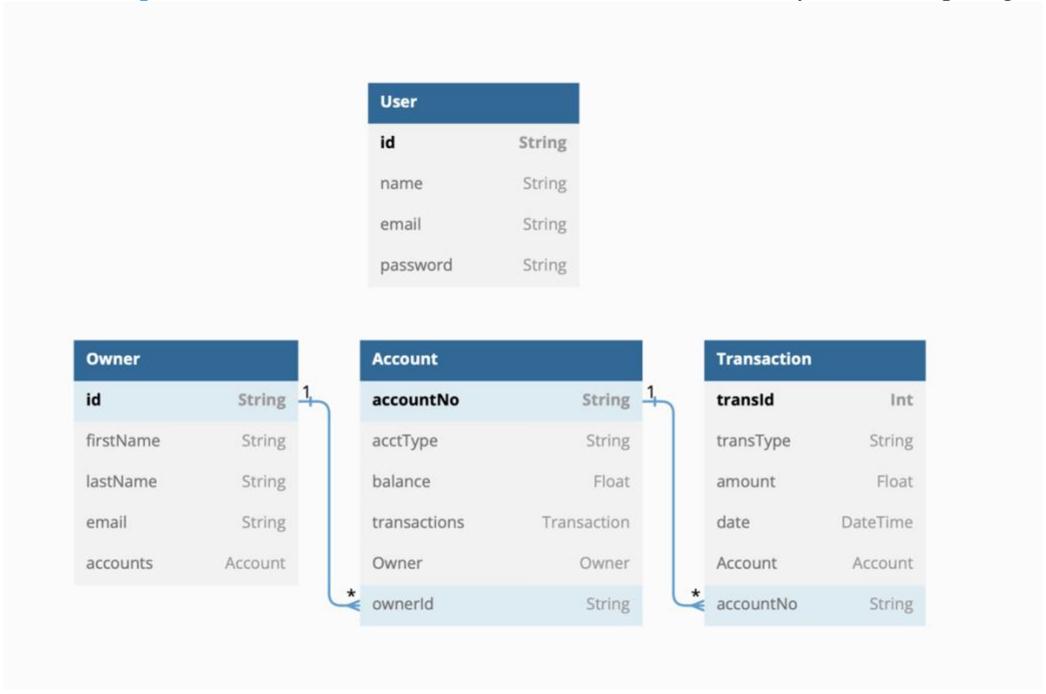
1. Run `npm install` to install the dependencies
2. create the **.env** file under the root of the project and add the following lines.

```
# The database URL for your Prisma service
DATABASE_URL="file:./dev.db"
#  The base URL of your application.
NEXTAUTH_URL='http://localhost:3000'
# The secret used to encrypt the cookie.   [you can use any random string here]
NEXTAUTH_SECRET= 'ASFLKJASDLKFJLASKJDFLKJASD '
# The secret used to sign the JWTs issued to clients. [you can use any random string here]
JWT_SECRET_KEY='ASFLKJASDLKFJLASKJDFLKJASD@'
```

3. `npx prisma db seed`
4. `npm run dev`

Update the Data Model

1. Modify the **schema.prisma** file and a new model named `user`. See the entity relationship diagram below.



2. Sync the models to your database by using the following Prisma command.
   `npx prisma migrate dev -name init`
3. To view your database content, run the following command `npx prisma studio`
4. Create a new file named `users-repo.js` under the `api/users` directory then implement the following repository functions:
   a. `addUser(user):` creates a new user. It will be used for user registration.
   b. `getUser(email, password):` returns the user object having the provided email and password. It will be used for user authentication.

Add Web API routes for registration and login

Add Web API routes for registration and login:

- **/api/users** – **POST** : for user registration
- **/api/users/login** – **POST** : for user authentication

Test the added routes using Postman.

Secure passwords using encryption

Secure user passwords stored in the database by encrypting them using **bcrypt** library.

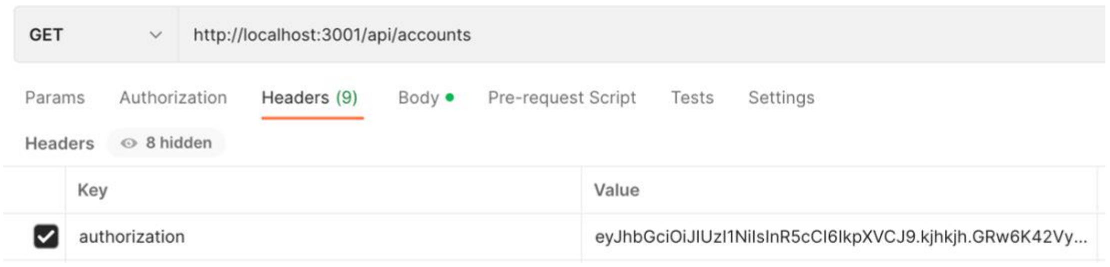
1. Install bcrypt using `npm i bcrypt` and then use it inside the `users-repo.js`

```
import * as bcrypt from 'bcrypt'

//inside the createUserfunction
// before adding the user we need to hash the password
const salt = await bcrypt.genSalt(10)
user.password = await bcrypt.hash(user.password, salt)

//inside the getUserfunction
const isMatch = await bcrypt.compare(password, user.password)
if (!isMatch) return { error: "Invalid credentials" }
```

Using JWT to authenticate and protect Web API routes

So far, we secured the passwords by encrypting them and we implemented basic authentication. Now we can extend the protection to secure Web API routes using **JWT Token** generated when the user is authenticate.

1. Install **JWT** token library `npm i jsonwebtoken`
2. Enhance authentication implemented by **/api/users/login** route. After successful user authentication generate a JWT `id_token` that represents the user's identity as a compact and **signed** string that can be easily transmitted to the client. as a means of authentication. JWT token is generated using the user object and the secret key.
   `const id_token = jwt.sign(user, process.env.JWT_SECRET, { expiresIn: '1d' })`
3. Now use the JWT Token to protect the Web API routes. To do this
   a. Read JWT token sent in the `authorization` header of the incoming request.
   b. If the authorization token is available, then verify that the token is valid using jwt library. Make sure to wrap the verification code with a try catch, as it will throw an exception in case the token is invalid.
      `const isValid jwt.verify(id_token, process.env.SECRET_KEY)`
   c. If the token is valid then allow the user to access the requested route, otherwise return `401 Unauthorized` error.
   d. Test you're your implementation using postman.


GET http://localhost:3001/api/accounts

	Key	Value
✓	authorization	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.kjhkjh.GRw6K42Vy...

Protecting the Web app pages and routes using NextAuth package

NextAuth.js is a flexible, easy to use and open-source authentication library for Next.js. It supports authentication using the traditional email/password authentication or by using multiple identity providers such as Facebook, Google, Twitter, Github.

1. Install NextAuth using `npm install next-auth`

2. To add **NextAuth** to a project create a **routes.js** under `/app/api/auth/[...nextauth]` folder. Then add the following code:

```
import NextAuth from "next-auth/next";
import CredentialsProvider from "next-auth/providers/credentials";

const handler = NextAuth({
    providers: [⟺]
});

export { handler as GET, handler as POST }
```

3. Configure NextAuth provider to generate a Login screen for traditional email/password authentication. Use the following link https://next-auth.js.org/providers/credentials for further details on how to configure the Username / Password Credentials provider.

```
export const authOptions = {
     providers: [
       CredentialsProvider({
         name: "Credentials",
      // `credentials` is used to generate a form on the sign in page.
      // You can specify which fields should be submitted, by adding keys
      // to the `credentials` object. e.g. email, password
         credentials: {
           email: {label: "Email", type: "text"},
           password: {label: "Password", type: "password"},
         },
      /* The function that next-auth calls to authenticate the user
        it takes the credentials submitted and returns a user object or null */
         async authorize(credentials, req) {

        const res = await fetch(`${process.env.NEXTAUTH_URL}/api/users/login`, {
             method: "POST",
             body: JSON.stringify(credentials),
             headers: { "Content-Type": "application/json" },
           })
           const user = await res.json()
           // If no error then return the user data otherwise return nulll
           if (res.ok && user) {
             return user
           }
           return null
         },
       }),
     ],
   }

const handler = NextAuth(authOptions)
// After configuring the next-auth handler, you can export it as
// GET and POST handlers for the /api/auth/[...nextauth] route
export { handler as GET, handler as POST }
```

Note the **authorize** method uses the **/api/users/login** Web API route you developed earlier.

4. To make the currently login-in user info available on the client-side to all pages create a new component named **UserInfoProvider.js,** which will have the following code

```
"use client"
import { SessionProvider } from "next-auth/react"
export default function UserInfoProvider({ children }) {
  return <SessionProvider>{children}</SessionProvider>
}
```

5. In the **layout.js file**, wrap the UI with the **UserInfoProvider** component. This step ensures that the currently login-in user info will be available on the client-side to the components within the application.

```
export default function RootLayout({ children }) {
  return (
    <html>
      <body>
        <UserInfoProvider>
          <NavBar />
          {children}
        </UserInfoProvider>
      </body>
    </html>
  )
}
```

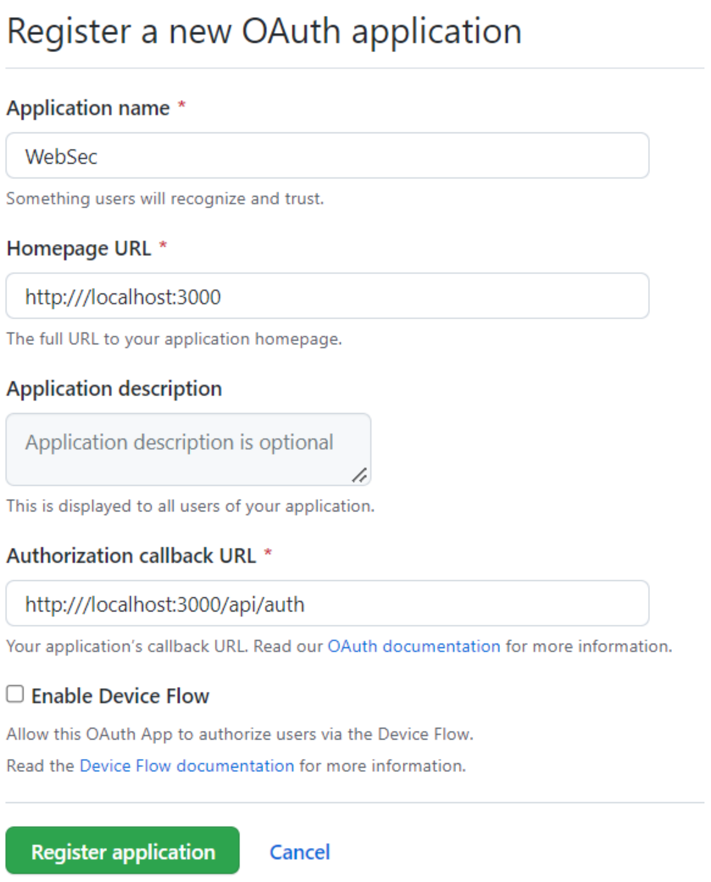
6. Test the app: `npm run dev`
   IMPORTANT: When running the application make sure it is running under PORT: 3000 OR it will not work. The reason for this is due to the value declared inside the .env file NEXTAUTH_URL=http://localhost:3000 .
7. Finally secure the different routes of the application, by creating a **middleware.js** file at the root of your application. Then add the following code:

```
export { default } from "next-auth/middleware"
export const config = {
    matcher: ['/accounts/:path*', '/api/accounts/:path*', '/api/owners/:path*']
}
```

Visiting /accounts, /api/accounts, /api/owners or nested routes (e.g., subpages like /accounts/123) will require authentication. If a user is not logged in the app will redirect them to the sign-in page.

NextAuth authentication using identity providers such as Google and Github

1. Create and configure GitHub OAuth Client using https://github.com/settings/developers


Register a new OAuth application

Application name *

WebSec

Something users will recognize and trust.

Homepage URL *

http://localhost:3000

The full URL to your application homepage.

Application description

Application description is optional

This is displayed to all users of your application.

Authorization callback URL *

http://localhost:3000/api/auth

Your application's callback URL. Read our OAuth documentation for more information.

☐ Enable Device Flow

Allow this OAuth App to authorize users via the Device Flow.

Read the Device Flow documentation for more information.

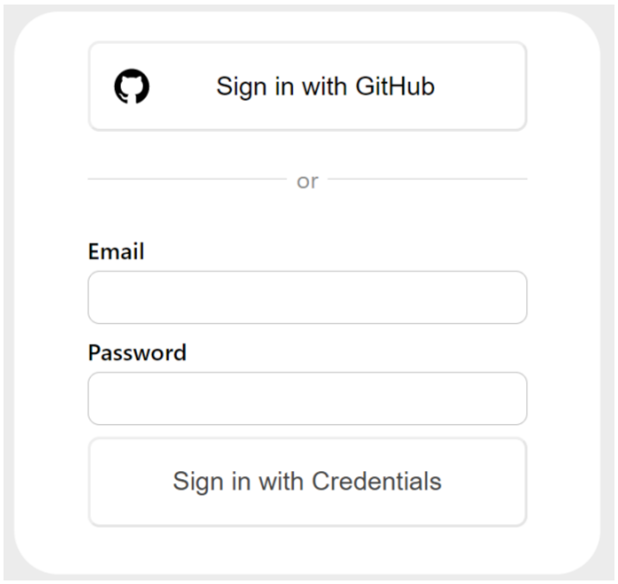
2. Copy the provided `Client ID` and `Client Secret` to `.env` file.

```
GITHUB_ID='...'
GITHUB_SECRET='...'
```

3. To add **GitHub provider** to the **routes.js** under `/app/api/auth/[...nextauth]` folder using the following code:

```
import GithubProvider from "next-auth/providers/github"

const handler = NextAuth({
  providers: [
    GithubProvider({
      clientId: process.env.GITHUB_ID,
      clientSecret: process.env.GITHUB_SECRET,
    }),

...
```

4. Test sign-in and you should get the ability to login using GitHub.



Using Custom Login and Registration

Instead of using the built-in **NextAuth** Library Login, we will use our own custom login to authenticate the user.

1. Open the […nextAuth] and add the following line at the bottom of the authOptions object

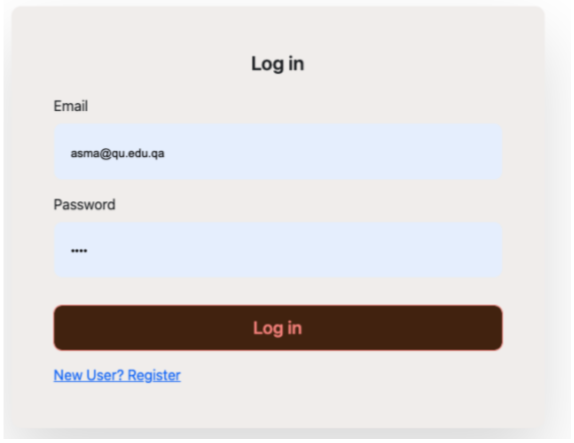
   pages : {

   signIn : '/login'

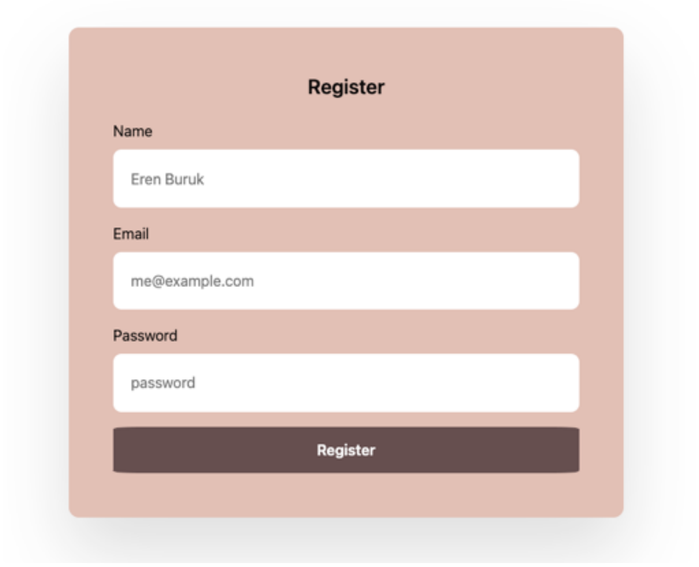
   }

2. Once the user submits the login form, then you should extract the email and password and call the **signIn** from method from **next-auth/react** with those parameters. Make sure you use server actions for the form submission.

```
import { signIn } from 'next-auth/react';
.....
....
await signIn('credentials', {
    email,
    password,
    callbackUrl: '/',
    redirect: true
})
```



3. Implement the register component using server actions


Register
Name
Eren Buruk
Email
me@example.com
Password
password
Register